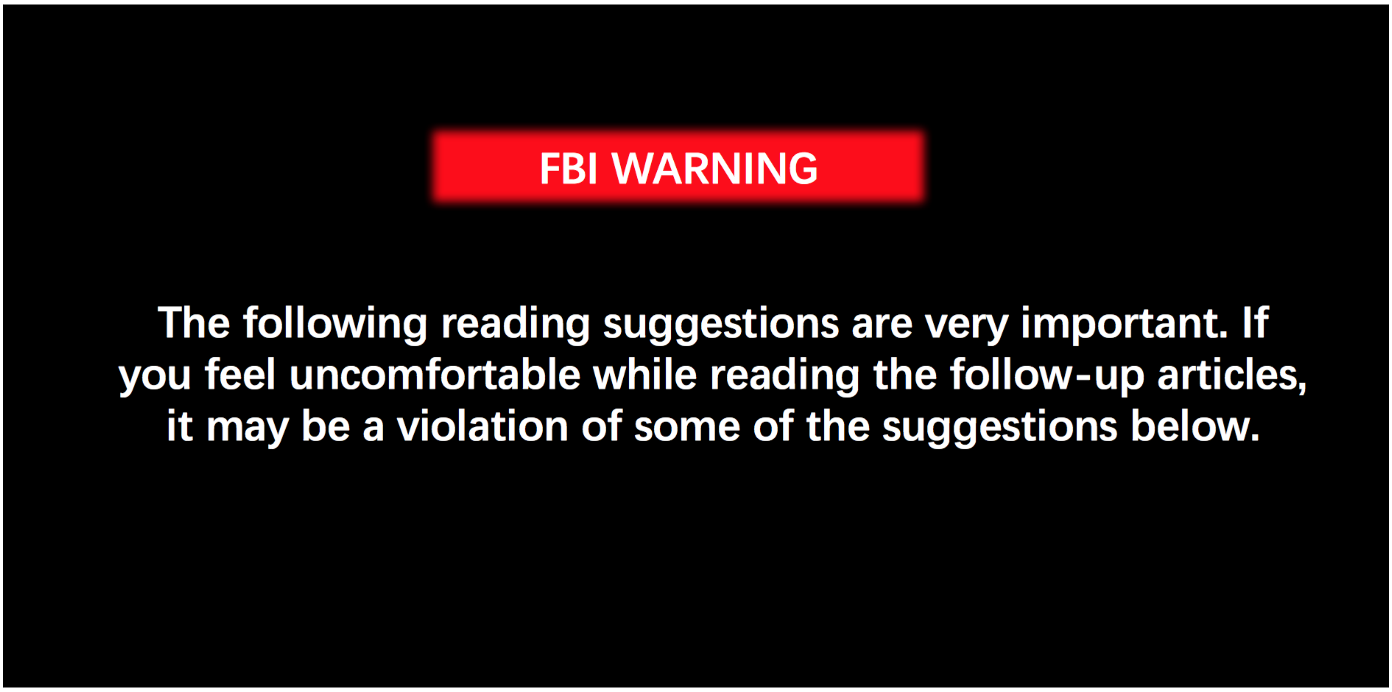
0 第0章 万里长征第一步（非常重要）-如何愉快的阅读本小册

0.1 购买前警告

- 此小册并非数据库入门书籍，需要各位知道增删改查是啥意思，并且能用 SQL 语言写出来，当然并不要求各位知道的太多，你甚至可以不知道连接的语法都可以。不过如果你连 `SELECT` 、 `INSERT` 这些单词都没听说过那本小册并不适合你。
- 此小册非正经科学专著，亦非十二五国家级规划教材，也没有大段代码和详细论证，有的全是图，喜欢正经论述的同学请避免购买本小册。
- 此小册作者乃一无业游民，非专业大佬，没有任何职称，只是单单喜欢把复杂问题讲清楚的那种快感，所以喜欢作者有 Google、Facebook 高级开发工程师，二百年工作经验等 Title 的同学请谨慎购买。
- 此小册是用于介绍 MySQL 的工作原理以及对我们程序猿的影响，并不是介绍概念设计、逻辑设计、物理设计、范式化之类的数据库设计方面的知识，希望了解上述这些知识的同学来错地方了。
- 文章标题中的“从根儿上理解MySQL”其实是专门雇了 UC 震惊部小编起的，纯属为了吸引大家眼球。严格意义上说，本书只是介绍 `MySQL` 内核的一些核心概念的小白进阶书籍。大家读完本小册也不会一下子晋升业界大佬，当上 CTO，迎娶白富美，走上人生巅峰。希望本小册能够帮助大家解决一些工作、面试过程中的问题，逐渐成为一个更好的工程师，有兴趣的小伙伴可以再深入研究一下 MySQL，说不定你就是下一个数据库泰斗啦。

0.2 购买并阅读本小册的建议



- 本小册是一本待出版的纸质书籍，并非一些杂碎文章的集合，是非常有结构和套路的，所以大家阅读时千万不能当作厕所蹲坑、吃饭看手机时的所谓 `碎片化读物` 。碎片化阅读只适合听听矮大紧、罗胖子他们扯扯犊子，开阔一下视野用的。对于专业的技术知识来说，大家必须付出一个完整的时间段进行体系化学习，这样尊重知识，工资才能尊重你。

> 顺便说一句，我已经好久都不听罗胖子扯犊子了，刚开始办罗辑思维的时候觉得他扯的还可以，越往后越觉得都钻钱眼儿里了，天天在鼓吹焦虑，让大家去买他们的鸡汤课。不过听听矮大紧就挺好啊，不累～

- 本小册是由 Markdown 写成，在电脑端阅读体验十分舒服，当然你非要用小手机看我也不拦着你，但是效果打了折扣是你的损失。
- 为了保证最好的阅读体验，不用一个没学过的概念去介绍另一个新概念，本小册的章节有严重的依赖性，比如你在没读 `InnoDB` 数据页结构前千万不要就去读 `B+` 树索引，所以大家最好从前看到尾，不要跳着看！不要跳着看！不要跳着看！，当然，不听劝告我也不能说啥，祝你好运。
- 大家可能买过别的小册，有的小册一篇文章可能用5分钟、10分钟读完，不过我的小册子每一篇文章都比较长，因为我把高耦合的部分都集中在一篇文章中了。文章中埋着各种伏笔，所以大家看的时候可能不会觉察出来很突兀的转变，所以在阅读一篇文章的时候千万不要跳着看！不要跳着看！不要跳着看！
- 大家在看本小册之前应该断断续续看过一些与本小册内容相关的知识，只是不成体系，细节学习的不够。对于这部分读者来说，希望大家像倚天屠龙记里的张无忌一样，在学张三丰的太极剑法时先忘记之前的武功，忘的越干净，学的越得真传。这样才能跟着我的套路走下去。
- 如果你真的是个小白的话，那这里头的数字都是假的：



一篇文章能用2个小时左右的时间掌握就很不错了。说句扫大家兴的话，虽然我已经很努力的想让大家的学习效率提升n倍，但是不幸的是想掌握一门核心技术仍然需要大家多看几遍（不然工资那么好涨啊～）。

0.3 关于工具

本小册中会涉及很多 InnoDB 的存储结构的知识，比如记录结构、页结构、索引结构、表空间结构等等，这些知识是所有后续知识的基础，所以是重中之重，需要大家认真对待。Jeremy Cole 已经使用 Ruby 开发了一个简易的解析这些基础结构的工具，github地址是：innodb_ruby的github地址(https://github.com/jeremycole/innodb_ruby)，大家可以按照说明安装上这个工具，可以更好的理解 InnoDB 中的一些存储结构（此工具虽然是针对 MySQL 5.6 的，但是幸好 MySQL 的基础存储结构基本没多大变化，所以大部分场景下这个 innodb_ruby 工具还是可以使用的）。

0.4 关于盗版

在写这本小册之前，我天真的以为只需要找几本参考书，看看 MySQL 的官方文档，遇到不会的地方百度谷歌一下就可以在 3 个月内解决这本书，后来的现实证明我真的想的太美了。不仅花了大量的时间阅读各种书籍和源码，而且有的时候知识耦合太厉害，为了更加模块化的把知识表述清楚，我又花了大量的时间来思考如何写作才能符合用户认知习惯，还花了非常多的时间来画各种图表，总之就是心累啊~

我希望的是：各位同学可以用很低的成本来更快速学会一些看起来生涩难懂的知识，但是毕竟我不是马云，不能一心一意做公益，希望各位通过正规渠道获得小册，尊重一下版权。

还有各位写博客的同学，引用的少了叫借鉴，引用的多了就，就有点那个了。希望各位不要大段大段的复制粘贴，用自己的话写出来的知识才是自己的东西。

我知道不论我们怎样强调版权意识，总是有一部分小伙伴喜欢不劳而获，总是喜欢想尽各种渠道来弄一份盗版的看，希望这部分同学看完之后记住能拍个大腿：这个叫小孩子的家伙写的真不错，之后在工作或者面试中用到了书里的东西还能想起我，当然，读完了之后记得关注一下公众号「我们都是小青蛙」。

> 小贴士：
>
> 我一直有个想法，就是如何降低教育成本。现在教育的盈利收费模式都太单一，就是直接跟学生收上课费，导致课程成为一种2C的商品，价格高低其实和内容质量并不是很相关，所以课程提供商会投入更大的精力做他们的渠道营销。所以现在的在线教育市场就是渠道为王，招生为王。我们其实可以换一种思路，在线教育的优势其实是传播费用更低，一个人上课和一千万人上课的费用区别其实就是服务器使用的多少罢了，所以我们可能并不需要那么多语文老师、数学老师，我们用专业的导演、专业的声优、专业的动画制作、专业的后期、专业的剪辑、专业的编剧组成的团队为某个科目制作一个专业的课程就好了嘛（顺便说一句，我就可以转行做课程编剧了）！把课程当作电影、电视剧来卖，只要在课程中植入广告，或者在播放平台上加广告就好了嘛，我们也可以在课程里培养偶像，来做一波粉丝经济。这样课程生产方也赚钱，学生们也省钱，最主要的是可以更大层度上促进教育公平，多好。

0.5 关于错误

0.5.1 准确性问题

我不是神，并不是书中的所有内容我都一一对照源码来验证准确性（阅读的大部分源码是关于查询优化和事务处理的），如果各位发现了文中有准确性问题请直接联系我，我会加入 Bug 列表中修正的。

0.5.2 阅读体验问题

大家知道大部分人在长大之后就忘记了自己小时候的样子，我写本书的初衷就是有很多资料我看不懂，看的我脑壳疼，之后才决定从小白的角度出发来写一本小白都能看懂的技术书籍。但是由于后来自己学的东西越来越多，可能有些地方我已经忘掉了小白的想法是怎么样的，所以大家在阅读过程中有任何阅读不畅快的地方都可以给我提，我也会加入bug列表中逐一优化。

0.6 关于转发

如果你从本小册中获取到了自己想要的知识，并且这个过程是比较轻松愉快的，希望各位能帮助转发本小册，解放一下学不懂这些知识的童鞋们，多节省一下他们的学习时间以及让学习过程不再那么痛苦。大家的技术都长进了，咱国家的技术也就慢慢强起来了。

0.7 关于疑惑

虽然我觉得文章写的已经很清晰了，但毕竟只是“我觉得”，不是大家觉得。传道授业解惑，解惑很重要。在学习一门知识时，我们最容易让一些问题绊住脚步，大家在阅读小册时如果发现了任何你觉得让你很困惑的问题，都可以直接加微信 xiaohaizi4919 问我，或者到群里提问题（最好到群里提，这样大家都能看到，也省的重复提问），我在力所能及的范围内尽力帮大家解答。



0.8 闲话

如果有的同学购买本小册后觉得并不是自己的菜，那很遗憾，我不能给你退款，钱是掘金这个平台收的。不过我还是觉得绝大部分同学读过后肯定有物超所值的感受，面试一般的数据库问题再也难不倒各位了，工作中一般的数据库问题也都是小菜一碟了，想继续研究 MySQL 源码的同学也找到方向了，如果你觉得 29.9 元不能表达你淘到宝的喜悦之情，那这好说，给我发红包就好了。

1 第1章 装作自己是个小白-重新认识MySQL

标签： MySQL是怎样运行的

1.1 MySQL的客户端／服务器架构

以我们平时使用的微信为例，它其实是由两部分组成的，一部分是客户端程序，一部分是服务器程序。客户端可能有很多种形式，比如手机APP，电脑软件或者是网页版微信，每个客户端都有一个唯一的用户名，就是你的微信号，另一方面，腾讯公司在他们的机房里运行着一个服务器软件，我们平时操作微信其实都是用客户端来和这个服务器来打交道。比如狗哥用微信给猫爷发了一条消息的过程其实是这样的：

1. 消息被客户端包装了一下，添加了发送者和接收者信息，然后从狗哥的微信客户端传送给微信服务器；
2. 微信服务器从消息里获取到它的发送者和接收者，根据消息的接收者信息把这条消息送达到猫爷的微信客户端，猫爷的微信客户端里就显示出狗哥给他发了一条消息。

`MySQL` 的使用过程跟这个是一样的，它的服务器程序直接和我们存储的数据打交道，然后可以有好多客户端程序连接到这个服务器程序，发送增删改查的请求，然后服务器就响应这些请求，从而操作它维护的数据。和微信一样， `MySQL` 的每个客户端都需要提供用户名密码才能登录，登录之后才能给服务器发请求来操作某些数据。我们

日常使用 MySQL 的情景一般是这样的：

1. 启动 MySQL 服务器程序。
2. 启动 MySQL 客户端程序并连接到服务器程序。
3. 在客户端程序中输入一些命令语句作为请求发送到服务器程序，服务器程序收到这些请求后，会根据请求的内容来操作具体的数据并向客户端返回操作结果。

我们知道计算机很牛逼，在一台计算机上可以同时运行多个程序，比如微信、QQ、音乐播放器、文本编辑器啥的，每一个运行着的程序也被称为一个 进程 。我们的 MySQL 服务器程序和客户端程序本质上都算是计算机上的一个 进程 ，这个代表着 MySQL 服务器程序的进程也被称为 MySQL数据库实例 ，简称 数据库实例 。

每个进程都有一个唯一的编号，称为 进程ID ，英文名叫 PID ，这个编号是在我们启动程序的时候由操作系统随机分配的，操作系统会保证在某一时刻同一台机器上的进程号不重复。比如你打开了计算机中的QQ程序，那么操作系统会为它分配一个唯一的进程号，如果你把这个程序关掉了，那操作系统就会把这个进程号回收，之后可能会重新分配给别的进程。当我们下一次再启动 QQ程序的时候分配的就可能是另一个编号。每个进程都有一个名称，这个名称是编写程序的人自己定义的，比如我们启动的 MySQL 服务器进程的默认名称为 mysqld ， 而我们常用的 MySQL 客户端进程的默认名称为 mysql 。

1.2 MySQL的安装

不论我们通过下载源代码自行编译安装的方式还是直接使用官方提供的安装包进行安装之后， MySQL 的服务器程序和客户端程序都会被安装到我们的机器上。不论使用上述两者的哪种安装方式，一定一定一定（重要的话说三遍）要记住你把 MySQL 安装到哪了，换句话说，一定要记住 MySQL 的安装目录。

> 小贴士：
> `MySQL`的大部分安装包都包含了服务器程序和客户端程序，不过在Linux下使用RPM包时会有单独的服务器RPM包和客户端RPM包，需要分别安装。

另外， MySQL 可以运行在各种各样的操作系统上，我们后边会讨论在类 UNIX 操作系统和 Windows 操作系统上使用的一些差别。为了方便大家理解，我在 macOS 操作系统（苹果电脑使用的操作系统）和 Windows 操作系统上都安装了 MySQL ，它们的安装目录分别是：

- macOS 操作系统上的安装目录：

  ```
  /usr/local/mysql/
  ```

- Windows 操作系统上的安装目录：

  ```
  C:\Program Files\MySQL\MySQL Server 5.7
  ```

  下边我会以这两个安装目录为例来进一步扯出更多的概念，不过一定要注意，**这两个安装目录是我的运行不同操作系统的机器上的安装目录，一定要记着把下边示例中用到安装目录的地方替换为你自己机器上的安装目录**。

  > 小贴士：
  > 类UNIX操作系统非常多，比如FreeBSD、Linux、macOS、Solaris等都属于UNIX操作系统的范畴，我们这里使用macOS操作系统代表类UNIX操作系统来运行MySQL。

1.2.1 bin目录下的可执行文件

在 MySQL 的安装目录下有一个特别特别重要的 bin 目录，这个目录下存放着许多可执行文件，以 macOS 系统为例，这个 bin 目录的绝对路径就是（在我的机器上）：

```
/usr/local/mysql/bin
```

我们列出一些在 macOS 中这个 bin 目录下的一部分可执行文件来看一下（文件太多，全列出来会刷屏的）：

```
.
├── mysql
├── mysql.server -> ../support-files/mysql.server
├── mysqladmin
├── mysqlbinlog
├── mysqlcheck
├── mysqld
├── mysqld_multi
├── mysqld_safe
├── mysqldump
├── mysqlimport
├── mysqlpump
... (省略其他文件)
0 directories, 40 files
```

`Windows` 中的可执行文件与 `macOS` 中的类似，不过都是以 `.exe` 为扩展名的。这些可执行文件都是与服务器程序和客户端程序相关的，后边我们会详细唠叨一些比较重要的可执行文件，现在先看看执行这些文件的方式。

对于有可视化界面的操作系统来说，我们拿着鼠标点点点就可以执行某个可执行文件，不过现在我们更关注在命令行环境下如何执行这些可执行文件，命令行通俗的说就是那些黑框框，这里的指的是类 `UNIX` 系统中的 `Shell` 或者 `Windows` 系统中的 `cmd.exe` ，如果你现在还不知道怎么启动这些命令行工具，网上搜搜吧~ 下边我们以 `macOS` 系统为例来看看如何启动这些可执行文件（ `Windows` 中的操作是类似的，依葫芦画瓢就好了）

- 使用可执行文件的相对/绝对路径 假设我们现在所处的工作目录是 `MySQL` 的安装目录，也就是 `/usr/local/mysql` ，我们想启动 `bin` 目录下的 `mysqld` 这个可执行文件，可以使用相对路径来启动：

  ```
  ./bin/mysqld
  ```

  或者直接输入 `mysqld` 的绝对路径也可以：

  ```
  /usr/local/mysql/bin/mysqld
  ```

- 将该 `bin` 目录的路径加入到环境变量 `PATH` 中

  如果我们觉得每次执行一个文件都要输入一串长长的路径名贼麻烦的话，可以把该 `bin` 目录所在的路径添加到环境变量 `PATH` 中。环境变量 `PATH` 是一系列路径的集合，各个路径之间使用冒号 `:` 隔离开，比方说我的机器上的环境变量 `PATH` 的值就是：

  ```
  /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
  ```

  我的系统中这个环境变量 `PATH` 的值表明：当我在输入一个命令时，系统便会在 `/usr/local/bin` 、 `/usr/bin:` 、 `/bin:` 、 `/usr/sbin` 、 `/sbin` 这些目录下依次寻找是否存在我们输入的那个命令，如果寻找成功，则执行该目录下对应的可执行文件。所以我们现在可以修改一下这个环境变量 `PATH` ，把 `MySQL` 安装目录下的 `bin` 目录的路径也加入到 `PATH` 中，在我的机器上修改后的环境变量 `PATH` 的值为：

  ```
  /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/mysql/bin
  ```

  这样现在不论我们所处的工作目录是啥，我们都可以直接输入可执行文件的名字就可以启动它，比如这样：

  ```
  mysqld
  ```

  方便多了哈~

小贴士：

关于啥是环境变量以及如何在当前系统中添加或修改系统变量不是我们唠叨的范围，大家找本相关的书或者上网查一查哈～

1.3 启动MySQL服务器程序

1.3.1 UNIX里启动服务器程序

在类 UNIX 系统中用来启动 MySQL 服务器程序的可执行文件有很多，大多在 MySQL 安装目录的 bin 目录下，我们一起来瞅瞅。

1.3.1.1 mysqld

mysqld 这个可执行文件就代表着 MySQL 服务器程序，运行这个可执行文件就可以直接启动一个服务器进程。但这个命令不常用，我们继续往下看更牛逼的启动命令。

1.3.1.2 mysqld_safe

mysqld_safe 是一个启动脚本，它会间接的调用 mysqld ，而且还顺便启动了另外一个监控进程，这个监控进程在服务器进程挂了的时候，可以帮助重启它。另外，使用 mysqld_safe 启动服务器程序时，它会将服务器程序的出错信息和其他诊断信息重定向到某个文件中，产生出错日志，这样可以方便我们找出发生错误的原因。

1.3.1.3 mysql.server

mysql.server 也是一个启动脚本，它会间接的调用 mysqld_safe ，在调用 mysql.server 时在后边指定 start 参数就可以启动服务器程序了，就像这样：

```
mysql.server start
```

需要注意的是，**这个 *mysql.server* 文件其实是一个链接文件，它的实际文件是 *../support-files/mysql.server***。我使用的 macOS 操作系统会帮我们在 bin 目录下自动创建一个指向实际文件的链接文件，如果你的操作系统没有帮你自动创建这个链接文件，那就自己创建一个呗～ 别告诉我你不会创建链接文件，上网搜搜呗～

另外，我们还可以使用 mysql.server 命令来关闭正在运行的服务器程序，只要把 start 参数换成 stop 就好了：

```
mysql.server stop
```

1.3.1.4 mysqld_multi

其实我们一台计算机上也可以运行多个服务器实例，也就是运行多个 MySQL 服务器进程。 mysql_multi 可执行文件可以对每一个服务器进程的启动或停止进行监控。这个命令的使用比较复杂，本书主要是为了讲清楚 MySQL 服务器和客户端运行的过程，不会对启动多个服务器程序进行过多唠叨。

1.3.2 Windows里启动服务器程序

Windows 里没有像类 UNIX 系统中那么多的启动脚本，但是也提供了手动启动和以服务的形式启动这两种方式，下边我们详细看。

1.3.2.1 mysqld

同样的，在 MySQL 安装目录下的 bin 目录下有一个 mysqld 可执行文件，在命令行里输入 mysqld ，或者直接双击运行它就算启动了 MySQL 服务器程序了。

1.3.2.2 以服务的方式运行服务器程序

首先看看啥是个 Windows 服务？如果无论是谁正在使用这台计算机，我们都需要长时间的运行某个程序，而且需要在计算机启动的时候便启动它，一般我们都会把它注册为一个 Windows 服务 ，操作系统会帮我们管理它。把某个程序注册为 Windows 服务的方式挺简单，如下：

```
"完整的可执行文件路径" --install [-manual] [服务名]
```

其中的 -manual 可以省略，加上它的话表示在 Windows 系统启动的时候不自动启动该服务，否则会自动启动。 服务名 也可以省略，默认的服务名就是 MySQL 。比如我的 Windows 计算机上 mysqld 的完整路径是：

```
C:\Program Files\MySQL\MySQL Server 5.7\bin\mysqld
```

所以如果我们想把它注册为服务的话可以在命令行里这么写：

```
"C:\Program Files\MySQL\MySQL Server 5.7\bin\mysqld" --install
```

在把 mysqld 注册为 Windows 服务之后，我们就可以通过下边这个命令来启动 MySQL 服务器程序了：

```
net start MySQL
```

当然，如果你喜欢图形界面的话，你可以通过 Windows 的服务管理器通过用鼠标点点点的方式来启动和停止服务（作为一个程序猿，还是用黑框框吧～）。

关闭这个服务也非常简单，只要把上边的 start 换成 stop 就行了，就像这样：

```
net stop MySQL
```

1.4 启动MySQL客户端程序

在我们成功启动 MySQL 服务器程序后，就可以接着启动客户端程序来连接到这个服务器喽， bin 目录下有许多客户端程序，比方说 mysqladmin 、 mysqldump 、 mysqlcheck 等等等等（好多呢，就不一一列举了）。这里我们重点要关注的是可执行文件 mysql ，通过这个可执行文件可以让我们和服务器程序进程交互，也就是发送请求，接收服务器的处理结果。启动这个可执行文件时一般需要一些参数，格式如下：

```
mysql -h主机名  -u用户名 -p密码
```

各个参数的意义如下： |参数名|含义| |:--:|:--:| | -h |表示服务器进程所在计算机的域名或者IP地址，如果服务器进程就运行在本机的话，可以省略这个参数，或者填 localhost 或者 127.0.0.1 。也可以写作 --host=主机名 的形式。| | -u |表示用户名。也可以写作 --user=用户名 的形式。| | -p |表示密码。也可以写作 --password=密码 的形式。|

> 小贴士：
>
> 像 h、u、p 这样名称只有一个英文字母的参数称为短形式的参数，使用时前边需要加单短划线，像 host、user、password 这样大于一个英文字母的参数称为长形式的参数，使用时前边需要加双短划线。
> 后边会详细讨论这些参数的使用方式的，稍安勿躁～

比如我这样执行下边这个可执行文件(用户名密码按你的实际情况填写)，就可以启动 MySQL 客户端，并且连接到服务器了。

```
mysql -hlocalhost -uroot -p123456
```

我们看一下连接成功后的界面：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.21 Homebrew

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

最后一行的 `mysql>` 是一个客户端的提示符，之后客户端发送给服务器的命令都需要写在这个提示符后边。

如果我们想断开客户端与服务器的连接并且关闭客户端的话，可以在 `mysql>` 提示符后输入下边任意一个命令：

1. `quit`
2. `exit`
3. `\q`

比如我们输入 `quit` 试试：

```
mysql> quit
Bye
```

输出了 `Bye` 说明客户端程序已经关掉了。注意注意注意，这是关闭客户端程序的方式，不是关闭服务器程序的方式，怎么关闭服务器程序上一节里唠叨过了。

如果你愿意，你可以多打开几个黑框框，每个黑框框都使用 `mysql -hlocahhost -uroot -p123456` 来运行多个客户端程序，每个客户端程序都是互不影响的。如果你有多个电脑，也可以试试把它们用局域网连起来，在一个电脑上启动 `MySQL` 服务器程序，在另一个电脑上执行 `mysql` 命令时使用 `IP` 地址作为主机名来连接到服务器。

1.4.1 连接注意事项

- 最好不要在一行命令中输入密码。

  我们直接在黑框框里输入密码很可能被别人看到，这和你当着别人的面输入银行卡密码没啥区别，所以我们在执行 `mysql` 连接服务器的时候可以不显式的写出密码，就像这样：

  ```
  mysql -hlocahhost -uroot -p
  ```

  点击回车之后才会提示你输入密码：

  ```
  Enter password:
  ```

  不过这回你输入的密码不会被显示出来，心怀不轨的人也就看不到了，输入完成点击回车就成功连接到了服务器。
- 如果你非要在一行命令中显式的把密码输出来，那 `-p` 和密码值之间不能有空白字符（其他参数名之间可以有空白字符），就像这样：

  ```
  mysql -h localhost -u root -p123456
  ```

  如果加上了空白字符就是错误的，比如这样：

  ```
  mysql -h localhost -u root -p 123456
  ```

- mysql 的各个参数的摆放顺序没有硬性规定，也就是说你也可以这么写：

```
mysql -p  -u root -h localhost
```

- 如果你的服务器和客户端安装在同一台机器上， -h 参数可以省略，就像这样：

```
mysql -u root -p
```

- 如果你使用的是类 UNIX 系统，并且省略 -u 参数后，会把你登陆操作系统的用户名当作 MySQL 的用户名去处理。

  比方说我用登录操作系统的用户名是 xiaohaizi ，那么在我的机器上下边这两条命令是等价的：

  ```
  mysql -u xiaohaizi -p
  mysql -p
  ```

  对于 Windows 系统来说，默认的用户名是 ODBC ，你可以通过设置环境变量 USER 来添加一个默认用户名。

1.5 客户端与服务器连接的过程

我们现在已经知道如何启动 MySQL 的服务器程序，以及如何启动客户端程序来连接到这个服务器程序。运行着的服务器程序和客户端程序本质上都是计算机上的一个进程，所以客户端进程向服务器进程发送请求并得到回复的过程本质上是一个进程间通信的过程！ MySQL 支持下边三种客户端进程和服务器进程的通信方式。

1.5.1 TCP/IP

真实环境中，数据库服务器进程和客户端进程可能运行在不同的主机中，它们之间必须通过网络来进行通讯。 MySQL 采用 TCP 作为服务器和客户端之间的网络通信协议。在网络环境下，每台计算机都有一个唯一的 IP地址 ，如果某个进程有需要采用 TCP 协议进行网络通信方面的需求，可以向操作系统申请一个 端口号 ，这是一个整数值，它的取值范围是 0~65535 。这样在网络中的其他进程就可以通过 IP地址 + 端口号 的方式来与这个进程连接，这样进程之间就可以通过网络进行通信了。

MySQL 服务器启动的时候会默认申请 3306 端口号，之后就在这个端口号上等待客户端进程进行连接，用书面一点的话来说， MySQL 服务器会默认监听 3306 端口。

```
小贴士：

`TCP/IP`网络体系结构是现在通用的一种网络体系结构，其中的`TCP`和`IP`是体系结构中两个非常重
要的网络协议，如果你并不知道协议是什么，或者并不知道网络是什么，那恐怕兄弟你来错地方了，找
本计算机网络的书去瞅瞅吧！

什么？计算机网络的书写的都贼晦涩，看不懂？没关系，等我~
```

如果 3306 端口号已经被别的进程占用了或者我们单纯的想自定义该数据库实例监听的端口号，那我们可以在启动服务器程序的命令行里添加 -P 参数来明确指定一下端口号，比如这样：

```
mysqld -P3307
```

这样 MySQL 服务器在启动时就会去监听我们指定的端口号 3307 。

如果客户端进程想要使用 TCP/IP 网络来连接到服务器进程，比如我们在使用 mysql 来启动客户端程序时，在 -h 参数后必须跟随 IP地址 来作为需要连接的服务器进程所在主机的主机名，如果客户端进程和服务器进程在一台计算机中的话，我们可以使用 127.0.0.1 来代表本机的 IP地址 。另外，如果服务器进程监听的端口号不是默认的 3306 ，我们也可以在使用 mysql 启动客户端程序时使用 -P 参数（大写的 P ，小写的 p 是用来指定密码的）来指定需要连接到的端口号。比如我们现在已经在本机启动了服务器进程，监听的端口号为 3307 ，那我们启动客户端程序时可以这样写：

```
mysql -h127.0.0.1 -uroot -P3307 -p
```

不知大家发现了没有，我们在启动服务器程序的命令 mysqld 和启动客户端程序的命令 mysql 后边都可以使用 -P 参数，关于如何在命令后边指定参数，指定哪些参数我们稍后会详细唠叨的，稍微等等哈～

1.5.2 命名管道和共享内存

如果你是一个 Windows 用户，那么客户端进程和服务器进程之间可以考虑使用 命名管道 或 共享内存 进行通信。不过启用这些通信方式的时候需要在启动服务器程序和客户端程序时添加一些参数：

- 使用 命名管道 来进行进程间通信

  需要在启动服务器程序的命令中加上 --enable-named-pipe 参数，然后在启动客户端程序的命令中加入 --pipe 或者 --protocol=pipe 参数。
- 使用 共享内存 来进行进程间通信

  需要在启动服务器程序的命令中加上 --shared-memory 参数，在成功启动服务器后， 共享内存 便成为本地客户端程序的默认连接方式，不过我们也可以在启动客户端程序的命令中加入 --protocol=memory 参数来显式的指定使用共享内存进行通信。

  不过需要注意的是，使用 共享内存 的方式进行通信的服务器进程和客户端进程必须在同一台 Windows 主机中。

> 小贴士：
> 命名管道和共享内存是Windows操作系统中的两种进程间通信方式，如果你没听过的话也不用纠结，并不妨碍我们介绍MySQL的知识～

1.5.3 Unix域套接字文件

如果我们的服务器进程和客户端进程都运行在同一台操作系统为类 Unix 的机器上的话，我们可以使用 Unix域套接字文件 来进行进程间通信。如果我们在启动客户端程序的时候指定的主机名为 localhost ，或者指定了 --protocol=socket 的启动参数，那服务器程序和客户端程序之间就可以通过 Unix 域套接字文件来进行通信了。 MySQL 服务器程序默认监听的 Unix 域套接字文件路径为 /tmp/mysql.sock ，客户端程序也默认连接到这个 Unix 域套接字文件。如果我们想改变这个默认路径，可以在启动服务器程序时指定 socket 参数，就像这样：

```
mysqld --socket=/tmp/a.txt
```

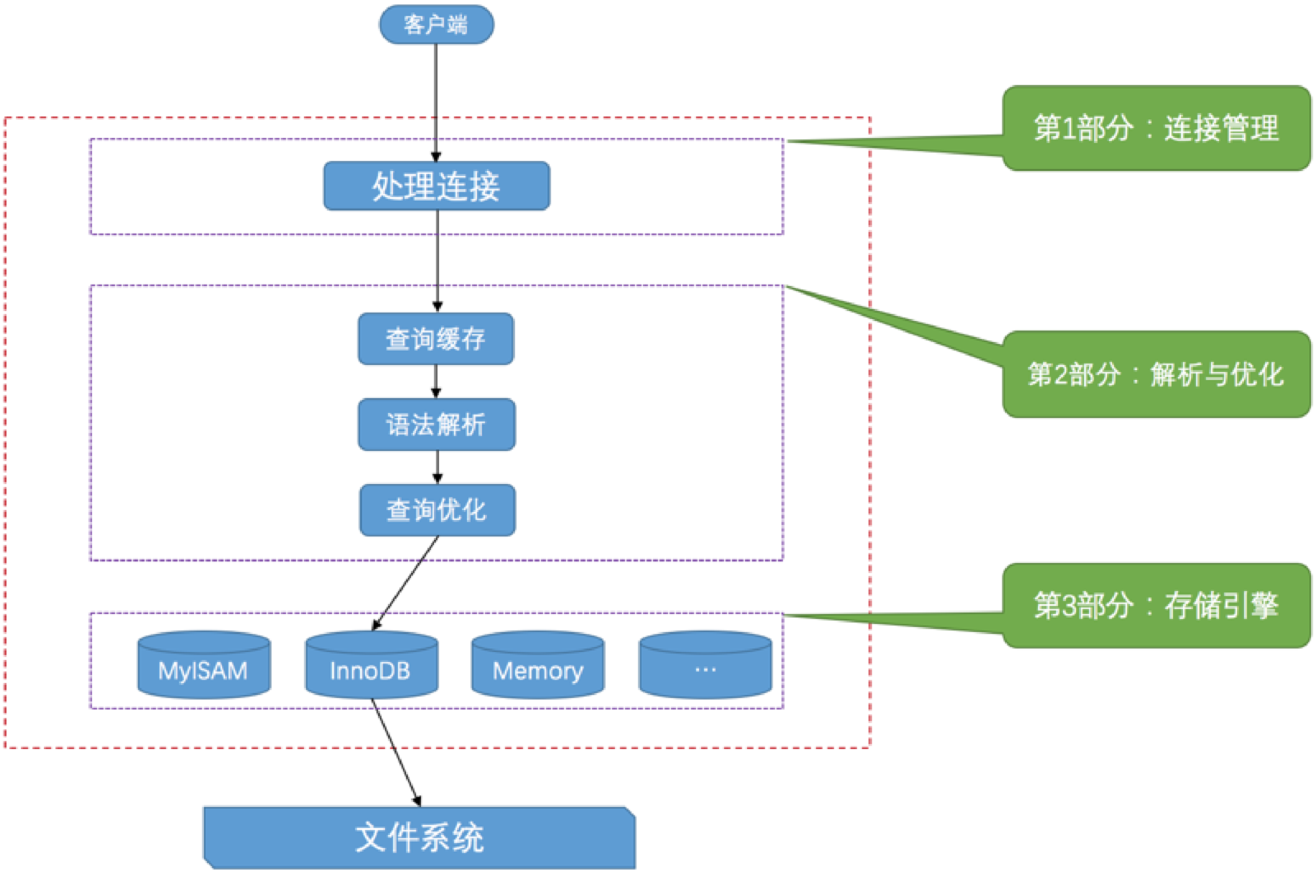
这样服务器启动后便会监听 /tmp/a.txt 。在服务器改变了默认的 UNIX 域套接字文件后，如果客户端程序想通过 UNIX 域套接字文件进行通信的话，也需要显式的指定连接到的 UNIX 域套接字文件路径，就像这样：

```
mysql -hlocalhost -uroot --socket=/tmp/a.txt -p
```

这样该客户端进程和服务器进程就可以通过路径为 /tmp/a.txt 的 Unix 域套接字文件进行通信了。

1.6 服务器处理客户端请求

其实不论客户端进程和服务器进程是采用哪种方式进行通信，最后实现的效果都是：客户端进程向服务器进程发送一段文本（MySQL语句），服务器进程处理后再向客户端进程发送一段文本（处理结果）。那服务器进程对客户端进程发送的请求做了什么处理，才能产生最后的处理结果呢？客户端可以向服务器发送增删改查各类请求，我们这里以比较复杂的查询请求为例来画个图展示一下大致的过程：



从图中我们可以看出，服务器程序处理来自客户端的查询请求大致需要经过三个部分，分别是 连接管理 、 解析与优化 、 存储引擎 。下边我们来详细看一下这三个部分都干了什么。

1.6.1 连接管理

客户端进程可以采用我们上边介绍的 TCP/IP 、 命名管道或共享内存 、 Unix域套接字 这几种方式之一来与服务器进程建立连接，每当有一个客户端进程连接到服务器进程时，服务器进程都会创建一个线程来专门处理与这个客户端的交互，当该客户端退出时会与服务器断开连接，服务器并不会立即把与该客户端交互的线程销毁掉，而是把它缓存起来，在另一个新的客户端再进行连接时，把这个缓存的线程分配给该新客户端。这样就起到了不频繁创建和销毁线程的效果，从而节省开销。从这一点大家也能看出， MySQL 服务器会为每一个连接进来的客户端分配一个线程，但是线程分配的太多了会严重影响系统性能，所以我们也需要限制一下可以同时连接到服务器的客户端数量，至于怎么限制我们后边再说哈～

在客户端程序发起连接的时候，需要携带主机信息、用户名、密码，服务器程序会对客户端程序提供的这些信息进行认证，如果认证失败，服务器程序会拒绝连接。另外，如果客户端程序和服务器程序不运行在一台计算机上，我们还可以采用使用了 SSL （安全套接字）的网络连接进行通信，来保证数据传输的安全性。

当连接建立后，与该客户端关联的服务器线程会一直等待客户端发送过来的请求， MySQL 服务器接收到的请求只是一个文本消息，该文本消息还要经过各种处理，预知后事如何，继续往下看哈～

1.6.2 解析与优化

到现在为止， MySQL 服务器已经获得了文本形式的请求，接着 还要经过九九八十一难的处理，其中的几个比较重要的部分分别是 查询缓存 、 语法解析 和 查询优化 ，下边我们详细来看。

1.6.2.1 查询缓存

如果我问你 9+8×16-3×2×17 的值是多少，你可能会用计算器去算一下，或者牛逼一点用心算，最终得到了结果 35 ，如果我再问你一遍 9+8×16-3×2×17 的值是多少，你还用再傻呵呵的算一遍么？我们刚刚已经算过了，直接说答案就好了。 MySQL 服务器程序处理查询请求的过程也是这样，会把刚刚处理过的查询请求和结果 缓存 起来，如果下一次有一模一样的请求过来，直接从缓存中查找结果就好了，就不用再傻呵呵的去底层的表中查找了。这个查询缓存可以在不同客户端之间共享，也就是说如果客户端A刚刚查询了一个语句，而客户端B之后发送了同样的查询请求，那么客户端B的这次查询就可以直接使用查询缓存中的数据。

当然， MySQL 服务器并没有人聪明，如果两个查询请求在任何字符上的不同（例如：空格、注释、大小写），都会导致缓存不会命中。另外，如果查询请求中包含某些系统函数、用户自定义变量和函数、一些系统表，如 mysql 、information_schema、 performance_schema 数据库中的表，那这个请求就不会被缓存。以某些系统函数举例，可能同样的函数的两次调用会产生不一样的结果，比如函数 NOW ，每次调用都会产生最新的当前时间，如果在一个查询请求中调用了这个函数，那即使查询请求的文本信息都一样，那不同时间的两次查询也应该得到不同的结果，如果在第一次查询时就缓存了，那第二次查询的时候直接使用第一次查询的结果就是错误的！

不过既然是缓存，那就有它缓存失效的时候。MySQL的缓存系统会监测涉及到的每张表，只要该表的结构或者数据被修改，如对该表使用了 INSERT 、 UPDATE 、 DELETE 、 TRUNCATE TABLE 、 ALTER TABLE 、 DROP TABLE 或 DROP DATABASE 语句，那使用该表的所有高速缓存查询都将变为无效并从高速缓存中删除！

> 小贴士：
> 虽然查询缓存有时可以提升系统性能，但也不得不因维护这块缓存而造成一些开销，比如每次都要去查询缓存中检索，查询请求处理完需要更新查询缓存，维护该查询缓存对应的内存区域。从MySQL 5.7.20开始，不推荐使用查询缓存，并在MySQL 8.0中删除。

1.6.2.2 语法解析

如果查询缓存没有命中，接下来就需要进入正式的查询阶段了。因为客户端程序发送过来的请求只是一段文本而已，所以 MySQL 服务器程序首先要对这段文本做分析，判断请求的语法是否正确，然后从文本中将要查询的表、各种查询条件都提取出来放到 MySQL 服务器内部使用的一些数据结构上来。

> 小贴士：
> 这个从指定的文本中提取出我们需要的信息本质上算是一个编译过程，涉及词法解析、语法分析、语义分析等阶段，这些问题不属于我们讨论的范畴，大家只要了解在处理请求的过程中需要这个步骤就好了。

1.6.2.3 查询优化

语法解析之后，服务器程序获得到了需要的信息，比如要查询的列是哪些，表是哪个，搜索条件是什么等等，但光有这些是不够的，因为我们写的 MySQL 语句执行起来效率可能并不是很高， MySQL 的优化程序会对我们的语句做一些优化，如外连接转换为内连接、表达式简化、子查询转为连接吧啦吧啦的一堆东西。优化的结果就是生成一个执行计划，这个执行计划表明了应该使用哪些索引进行查询，表之间的连接顺序是啥样的。我们可以使用 EXPLAIN 语句来查看某个语句的执行计划，关于查询优化这部分的详细内容我们后边会仔细唠叨，现在你只需要知道在 MySQL 服务器程序处理请求的过程中有这么一个步骤就好了。

1.6.3 存储引擎

截止到服务器程序完成了查询优化为止，还没有真正的去访问真实的数据表， MySQL 服务器把数据的存储和提取操作都封装到了一个叫 存储引擎 的模块里。我们知道 表 是由一行一行的记录组成的，但这只是一个逻辑上的概念，物理上如何表示记录，怎么从表中读取数据，怎么把数据写入具体的物理存储器上，这都是 存储引擎 负责的事情。为了实现不同的功能， MySQL 提供了各式各样的 存储引擎 ，不同 存储引擎 管理的表具体的存储结构可能不同，采用的存取算法也可能不同。

小贴士：
为什么叫`引擎`呢？因为这个名字更拉风～ 其实这个存储引擎以前叫做`表处理器`，后来可能人们觉得太土，就改成了`存储引擎`的叫法，它的功能就是接收上层传下来的指令，然后对表中的数据进行提取或写入操作。

为了管理方便，人们把 连接管理 、 查询缓存 、 语法解析 、 查询优化 这些并不涉及真实数据存储的功能划分为 MySQL server 的功能，把真实存取数据的功能划分为 存储引擎 的功能。各种不同的存储引擎向上边的 MySQL server 层提供统一的调用接口（也就是存储引擎API），包含了几十个底层函数，像"读取索引第一条内容"、"读取索引下一条内容"、"插入记录"等等。

所以在 MySQL server 完成了查询优化后，只需按照生成的执行计划调用底层存储引擎提供的API，获取到数据后返回给客户端就好了。

1.7 常用存储引擎

MySQL 支持非常多种存储引擎，我这先列举一些：

存储引擎	描述
ARCHIVE	用于数据存档（行被插入后不能再修改）
BLACKHOLE	丢弃写操作，读操作会返回空内容
CSV	在存储数据时，以逗号分隔各个数据项
FEDERATED	用来访问远程表
InnoDB	具备外键支持功能的事务存储引擎
MEMORY	置于内存的表
MERGE	用来管理多个MyISAM表构成的表集合
MyISAM	主要的非事务处理存储引擎
NDB	MySQL集群专用存储引擎

这么多我们怎么挑啊，哈哈，你多虑了，其实我们最常用的就是 InnoDB 和 MyISAM ，有时会提一下 Memory 。其中 InnoDB 是 MySQL 默认的存储引擎，我们之后会详细唠叨这个存储引擎的各种功能，现在先看一下一些存储引擎对于某些功能的支持情况：

Feature	MyISAM	Memory	InnoDB	Archive	NDB
B-tree indexes	yes	yes	yes	no	no
Backup/point-in-time recovery	yes	yes	yes	yes	yes
Cluster database support	no	no	no	no	yes
Clustered indexes	no	no	yes	no	no
Compressed data	yes	no	yes	yes	no
Data caches	no	N/A	yes	no	yes
Encrypted data	yes	yes	yes	yes	yes
Foreign key support	no	no	yes	no	yes
Full-text search indexes	yes	no	yes	no	no
Geospatial data type support	yes	no	yes	yes	yes
Geospatial indexing support	yes	no	yes	no	no
Hash indexes	no	yes	no	no	yes

Feature	MyISAM	Memory	InnoDB	Archive	NDB
Index caches	yes	N/A	yes	no	yes
Locking granularity	Table	Table	Row	Row	Row
MVCC	no	no	yes	no	no
Query cache support	yes	yes	yes	yes	yes
Replication support	yes	Limited	yes	yes	yes
Storage limits	256TB	RAM	64TB	None	384EB
T-tree indexes	no	no	no	no	yes
Transactions	no	no	yes	no	yes
Update statistics for data dictionary	yes	yes	yes	yes	yes

密密麻麻列了这么多，看的头皮都发麻了，达到的效果就是告诉你：这玩意儿很复杂。其实这些东西大家没必要立即就给记住，我列出来的目的就是想让大家明白不同的存储引擎支持不同的功能，有些重要的功能我们会在后边的唠叨中慢慢让大家理解的～

1.8 关于存储引擎的一些操作

1.8.1 查看当前服务器程序支持的存储引擎

我们可以用下边这个命令来查看当前服务器程序支持的存储引擎：

```
SHOW ENGINES;
```

来看一下调用效果：

```
mysql> SHOW ENGINES;
+--------------------+---------+----------------------------------------------------------------
------+--------------+------+------------+
| Engine             | Support | Comment
| Transactions | XA   | Savepoints |
+--------------------+---------+----------------------------------------------------------------
------+--------------+------+------------+
| InnoDB             | DEFAULT | Supports transactions, row-level locking, and foreign key
s     | YES          | YES  | YES        |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables
| NO           | NO   | NO         |
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary tables
| NO           | NO   | NO         |
| BLACKHOLE          | YES     | /dev/null storage engine (anything you write to it disapp
ears) | NO           | NO   | NO         |
| MyISAM             | YES     | MyISAM storage engine
| NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine
| NO           | NO   | NO         |
| ARCHIVE            | YES     | Archive storage engine
| NO           | NO   | NO         |
| PERFORMANCE_SCHEMA | YES     | Performance Schema
| NO           | NO   | NO         |
| FEDERATED          | NO      | Federated MySQL storage engine
| NULL         | NULL | NULL       |
+--------------------+---------+----------------------------------------------------------------
------+--------------+------+------------+
9 rows in set (0.00 sec)

mysql>
```

其中的 Support 列表示该存储引擎是否可用， DEFAULT 值代表是当前服务器程序的默认存储引擎。 Comment 列是对存储引擎的一个描述，英文的，将就着看吧。 Transactions 列代表该存储引擎是否支持事务处理。 XA 列代表着该存储引擎是否支持分布式事务。 Savepoints 代表着该列是否支持部分事务回滚。

> 小贴士：
> 好吧，也许你并不知道什么是个事务、更别提分布式事务了，这些内容我们在后边的章节会详细唠叨，现在瞅一眼看个新鲜就得了。

1.8.2 设置表的存储引擎

我们前边说过，存储引擎是负责对表中的数据进行提取和写入工作的，我们可以为不同的表设置不同的存储引擎，也就是说不同的表可以有不同的物理存储结构，不同的提取和写入方式。

1.8.2.1 创建表时指定存储引擎

我们之前创建表的语句都没有指定表的存储引擎，那就会使用默认的存储引擎 InnoDB （当然这个默认的存储引擎也是可以修改的，我们在后边的章节中再说怎么改）。如果我们想显式的指定一下表的存储引擎，那可以这么写：

```
CREATE TABLE 表名(
    建表语句;
) ENGINE = 存储引擎名称;
```

比如我们想创建一个存储引擎为 MyISAM 的表可以这么写：

```
mysql> CREATE TABLE engine_demo_table(
    ->     i int
    -> ) ENGINE = MyISAM;
Query OK, 0 rows affected (0.02 sec)

mysql>
```

1.8.2.2 修改表的存储引擎

如果表已经建好了，我们也可以使用下边这个语句来修改表的存储引擎：

```
ALTER TABLE 表名 ENGINE = 存储引擎名称;
```

比如我们修改一下 engine_demo_table 表的存储引擎：

```
mysql> ALTER TABLE engine_demo_table ENGINE = InnoDB;
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
```

这时我们再查看一下 engine_demo_table 的表结构：

```
mysql> SHOW CREATE TABLE engine_demo_table\G
*************************** 1. row ***************************
       Table: engine_demo_table
Create Table: CREATE TABLE `engine_demo_table` (
  `i` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.01 sec)

mysql>
```

可以看到该表的存储引擎已经改为 InnoDB 了。

2 第2章 MySQL的调控按钮-启动选项和系统变量

标签：MySQL 是怎样运行的

如果你用过手机，你的手机上一定有一个设置的功能，你可以选择设置手机的来电铃声、设置音量大小、设置解锁密码等等。假如没有这些设置功能，我们的生活将置于尴尬的境地，比如在图书馆里无法把手机设置为静音，无法把流量开关关掉以节省流量，在别人得知解锁密码后无法更改密码～ MySQL 的服务器程序和客户端程序也有很多设置项，比如对于 MySQL 服务器程序，我们可以指定诸如允许同时连入的客户端数量、客户端和服务器通信方式、表的默认存储引擎、查询缓存的大小吧啦吧啦的设置项。对于 MySQL 客户端程序，我们之前已经见识过了，可以指定需要连接的服务器程序所在主机的主机名或IP地址、用户名及密码等信息。

这些设置项一般都有各自的默认值，比方说服务器允许同时连入的客户端的默认数量是 151 ，表的默认存储引擎是 InnoDB ，我们可以在程序启动的时候去修改这些默认值，对于这种在程序启动时指定的设置项也称之为启动选项（startup options），这些选项控制着程序启动后的行为。在 MySQL 安装目录下的 bin 目录中的各种可执行文件，不论是服务器相关的程序（比如 mysqld 、 mysqld_safe ）还是客户端相关的程序（比如 mysql 、 mysqladmin ），在启动的时候基本都可以指定启动参数。这些启动参数可以放在命令行中指定，也可以把它们

放在配置文件中指定。下边我们会以 mysqld 为例，来详细唠叨指定启动选项的格式。需要注意的一点是，我们现在要唠叨的是设置启动选项的方式，下边出现的启动选项不论大家认不认识，先不用去纠结每个选项具体的作用是啥，之后我们会对一些重要的启动选项详细唠叨。

2.1 在命令行上使用选项

如果我们在启动客户端程序时在 -h 参数后边紧跟服务器的IP地址，这就意味着客户端和服务器之间需要通过 TCP/IP 网络进行通信。因为我的客户端程序和服务器程序都装在一台计算机上，所以在使用客户端程序连接服务器程序时指定的主机名是 127.0.0.1 的情况下，客户端进程和服务器进程之间会使用 TCP/IP 网络进行通信。如果我们在启动服务器程序的时候就禁止各客户端使用 TCP/IP 网络进行通信，可以在启动服务器程序的命令行里添加 skip-networking 启动选项，就像这样：

```
mysqld --skip-networking
```

可以看到，我们在命令行中指定启动选项时需要在选项名前加上 -- 前缀。另外，如果选项名是由多个单词构成的，它们之间可以由短划线 - 连接起来，也可以使用下划线 _ 连接起来，也就是说 skip-networking 和 skip_networking 表示的含义是相同的。所以上边的写法与下边的写法是等价的：

```
mysqld --skip_networking
```

在按照上述命令启动服务器程序后，如果我们再使用 mysql 来启动客户端程序时，再把服务器主机名指定为 127.0.0.1 （IP地址的形式）的话会显示连接失败：

```
 mysql -h127.0.0.1 -uroot -p
Enter password:

ERROR 2003 (HY000): Can't connect to MySQL server on '127.0.0.1' (61)
```

这就意味着我们指定的启动选项 skip-networking 生效了！

再举一个例子，我们前边说过如果在创建表的语句中没有显式指定表的存储引擎的话，那就会默认使用 InnoDB 作为表的存储引擎。如果我们想改变表的默认存储引擎的话，可以这样写启动服务器的命令行：

```
mysqld --default-storage-engine=MyISAM
```

我们现在就已经把表的默认存储引擎改为 MyISAM 了，在客户端程序连接到服务器程序后试着创建一个表：

```
mysql> CREATE TABLE sys_var_demo(
    ->     i INT
    -> );
Query OK, 0 rows affected (0.02 sec)
```

这个定义语句中我们并没有明确指定表的存储引擎，创建成功后再看一下这个表的结构：

```
mysql> SHOW CREATE TABLE sys_var_demo\G
*************************** 1. row ***************************
       Table: sys_var_demo
Create Table: CREATE TABLE `sys_var_demo` (
  `i` int(11) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8
1 row in set (0.01 sec)
```

可以看到该表的存储引擎已经是 MyISAM 了，说明启动选项 default-storage-engine 生效了。

所以在启动服务器程序的命令行后边指定启动选项的通用格式就是这样的：

```
--启动选项1[=值1] --启动选项2[=值2] ... --启动选项n[=值n]
```

也就是说我们可以将各个启动选项写到一行中，各个启动选项之间使用空白字符隔开，在每一个启动选项名称前边添加 -- 。对于不需要值的启动选项，比方说 skip-networking ，它们就不需要指定对应的值。对于需要指定值的启动选项，比如 default-storage-engine 我们在指定这个设置项的时候需要显式的指定它的值，比方说 InnoDB 、 MyISAM 啦什么的~ 在命令行上指定有值的启动选项时需要注意，选项名、=、选项值之间不可以有空白字符，比如写成下边这样就是不正确的：

```
mysqld --default-storage-engine = MyISAM
```

每个MySQL程序都有许多不同的选项。大多数程序提供了一个--help选项，你可以查看该程序支持的全部启动选项以及它们的默认值。例如，使用 mysql --help 可以看到 mysql 程序支持的启动选项， mysqld_safe --help 可以看到 mysqld_safe 程序支持的启动选项。查看 mysqld 支持的启动选项有些特别，需要使用 mysqld --verbose --help 。

2.1.1 选项的长形式和短形式

我们前边提到的 skip-networking 、 default-storage-engine 称之为长形式的选项（因为它们很长），设计 MySQL 的大叔为了我们使用的方便，对于一些常用的选项提供了短形式，我们列举一些具有短形式的启动选项来瞅瞅（ MySQL 支持的短形式选项太多了，全列出来会刷屏的）：

长形式	短形式	含义
--host	-h	主机名
--user	-u	用户名
--password	-p	密码
--port	-P	端口
--version	-V	版本信息

短形式的选项名只有一个字母，与使用长形式选项时需要在选项名前加两个短划线 -- 不同的是，使用短形式选项时在选项名前只加一个短划线 - 前缀。有一些短形式的选项我们之前已经接触过了，比方说我们在启动服务器程序时指定监听的端口号：

```
mysqld -P3307
```

使用短形式指定启动选项时，选项名和选项值之间可以没有间隙，或者用空白字符隔开（ -p 选项有些特殊， -p 和密码值之间不能有空白字符），也就是说上边的命令形式和下边的是等价的：

```
mysqld -P 3307
```

另外，选项名是区分大小写的，比如 -p 和 -P 选项拥有完全不同的含义，大家需要注意一下。

2.2 配置文件中使用选项

在命令行中设置启动选项只对当次启动生效，也就是说如果下一次重启程序的时候我们还想保留这些启动选项的话，还得重复把这些选项写到启动命令行中，这样真的神烦唉！于是设计 MySQL 的大叔们提出一种 配置文件 （也称为 选项文件 ）的概念，我们把需要设置的启动选项都写在这个配置文件中，每次启动服务器的时候都从这个文件里加载相应的启动选项。由于这个配置文件可以长久的保存在计算机的硬盘里，所以只需我们配置一次，以后就都不用显式的把启动选项都写在启动命令行中了，所以我们推荐使用配置文件的方式来设置启动选项。

2.2.1 配置文件的路径

MySQL 程序在启动时会寻找多个路径下的配置文件，这些路径有的是固定的，有的是可以在命令行指定的。根据操作系统的不同，配置文件的路径也有所不同，我们分开看一下。

2.2.1.1 Windows操作系统的配置文件

在 Windows 操作系统中， MySQL 会按照下列路径来寻找配置文件：

路径名	备注
%WINDIR%\my.ini ， %WINDIR%\my.cnf	
C:\my.ini ， C:\my.cnf	
BASEDIR\my.ini ， BASEDIR\my.cnf	
defaults-extra-file	命令行指定的额外配置文件路径
%APPDATA%\MySQL\.mylogin.cnf	登录路径选项（仅限客户端）

在阅读这些 Windows 操作系统下配置文件路径的时候需要注意一些事情：

- 在给定的前三个路径中，配置文件可以使用 .ini 的扩展名，也可以使用 .cnf 的扩展名。
- %WINDIR% 指的是你机器上 Windows 目录的位置，通常是 C:\WINDOWS ，如果你不确定，可以使用这个命令来查看：

  ```
  echo %WINDIR%
  ```

- BASEDIR 指的是 MySQL 安装目录的路径，在我的 Windows 机器上的 BASEDIR 的值是：

  ```
  C:\Program Files\MySQL\MySQL Server 5.7
  ```

- 第四个路径指的是我们在启动程序时可以通过指定 defaults-extra-file 参数的值来添加额外的配置文件路径，比方说我们在命令行上可以这么写：

  ```
  mysqld --defaults-extra-file=C:\Users\xiaohaizi\my_extra_file.txt
  ```

  这样 MySQL 服务器启动时就可以额外在 C:\Users\xiaohaizi\my_extra_file.txt 这个路径下查找配置文件。
- %APPDATA% 表示 Windows 应用程序数据目录的值，可以使用下列命令查看：

  ```
  echo %APPDATA%
  ```

- 列表中最后一个名为 .mylogin.cnf 配置文件有点儿特殊，它不是一个纯文本文件（其他的配置文件都是纯文本文件），而是使用 mysql_config_editor 实用程序创建的加密文件。文件中只能包含一些用于启动客户端软件时连接服务器的一些选项，包括 host 、 user 、 password 、 port 和 socket 。而且它只能被客户端程序所使用。

  小贴士：
  mysql_config_editor实用程序其实是MySQL安装目录下的bin目录下的一个可执行文件，这个实用程序有专用的语法来生成或修改 .mylogin.cnf 文件中的内容，如何使用这个程序不是我们讨论的主题，可以到MySQL的官方文档中查看。

2.2.1.2 类Unix操作系统中的配置文件

在类 UNIX 操作系统中， MySQL 会按照下列路径来寻找配置文件：

路径名	备注
/etc/my.cnf	

路径名	备注
/etc/mysql/my.cnf	
SYSCONFDIR/my.cnf	
$MYSQL_HOME/my.cnf	特定于服务器的选项（仅限服务器）
defaults-extra-file	命令行指定的额外配置文件路径
~/.my.cnf	用户特定选项
~/.mylogin.cnf	用户特定的登录路径选项（仅限客户端）

在阅读这些 UNIX 操作系统下配置文件路径的时候需要注意一些事情：

- SYSCONFDIR 表示在使用 CMake 构建 MySQL 时使用 SYSCONFDIR 选项指定的目录。默认情况下，这是位于编译安装目录下的 etc 目录。

  > 小贴士：
  > 如果你不懂啥是个CMAKE，啥是个编译，那就跳过吧，对我们后续的文章没啥影响。

- MYSQL_HOME 是一个环境变量，该变量的值是我们自己设置的，我们想设置就设置，不想设置就不设置。该变量的值代表一个路径，我们可以在该路径下创建一个 my.cnf 配置文件，那么这个配置文件中只能放置关于启动服务器程序相关的选项（言外之意就是其他的配置文件既能存放服务器相关的选项也能存放客户端相关的选项， .mylogin.cnf 除外，它只能存放客户端相关的一些选项）。

  > 小贴士：
  > 如果大家使用mysqld_safe启动服务器程序，而且我们也没有主动设置这个MySQL_HOME环境变量的值，那这个环境变量的值将自动被设置为MySQL的安装目录，也就是MySQL服务器将会在安装目录下查找名为my.cnf配置文件（别忘了mysql.server会调用mysqld_safe，所以使用mysql.server启动服务器时也会在安装目录下查找配置文件）。

- 列表中的最后两个以 ~ 开头的路径是用户相关的，类 UNIX 系统中都有一个当前登陆用户的概念，每个用户都可以有一个用户目录， ~ 就代表这个用户目录，大家可以查看 HOME 环境变量的值来确定一下当前用户的用户目录，比方说我的 macOS 机器上的用户目录就是 /Users/xiaohaizi 。之所以说列表中最后两个配置文件是用户相关的，是因为不同的类 UNIX 系统的用户都可以在自己的用户目录下创建 .my.cnf 或者 .mylogin.cnf ，换句话说，不同登录用户使用的 .my.cnf 或者 .mylogin.cnf 配置文件是不同的。
- defaults-extra-file 的含义与Windows中的一样。
- .mylogin.cnf 的含义也同 Windows 中的一样，再次强调一遍，它不是纯文本文件，只能使用 mysql_config_editor 实用程序去创建或修改，用于存放客户端登陆服务器时的相关选项。

这也就是说，在我的计算机中这几个路径中的任意一个都可以当作配置文件来使用，如果它们不存在，你可以手动创建一个，比方说我手动在 ~/.my.cnf 这个路径下创建一个配置文件。

另外，我们在唠叨如何启动 MySQL 服务器程序的时候说过，使用 mysqld_safe 程序启动服务器时，会间接调用 mysqld ，所以对于传递给 mysqld_safe 的启动选项来说，如果 mysqld_safe 程序不处理，会接着传递给 mysqld 程序处理。比方说 skip-networking 选项是由 mysqld 处理的， mysqld_safe 并不处理，但是如果我们我们在命令行上这样执行：

```
mysqld_safe --skip-networking
```

则在 mysqld_safe 调用 mysqld 时，会把它处理不了的这个 skip-networking 选项交给 mysqld 处理。

2.2.2 配置文件的内容

与在命令行中指定启动选项不同的是，配置文件中的启动选项被划分为若干个组，每个组有一个组名，用中括号 [] 扩起来，像这样：

```
[server]
(具体的启动选项...)

[mysqld]
(具体的启动选项...)

[mysqld_safe]
(具体的启动选项...)

[client]
(具体的启动选项...)

[mysql]
(具体的启动选项...)

[mysqladmin]
(具体的启动选项...)
```

像这个配置文件里就定义了许多个组，组名分别是 server 、 mysqld 、 mysqld_safe 、 client 、 mysql 、 mysqladmin 。每个组下边可以定义若干个启动选项，我们以 [server] 组为例来看一下填写启动选项的形式（其他组中启动选项的形式是一样的）：

```
[server]
option1     #这是option1，该选项不需要选项值
option2 = value2      #这是option2，该选项需要选项值
...
```

在配置文件中指定启动选项的语法类似于命令行语法，但是配置文件中只能使用长形式的选项。在配置文件中指定的启动选项不允许加 -- 前缀，并且每行只指定一个选项，而且 = 周围可以有空白字符（命令行中选项名、 = 、选项值之间不允许有空白字符）。另外，在配置文件中，我们可以使用 # 来添加注释，从 # 出现直到行尾的内容都属于注释内容，读取配置文件时会忽略这些注释内容。为了大家更容易对比启动选项在命令行和配置文件中指定的区别，我们再把命令行中指定 option1 和 option2 两个选项的格式写一遍看看：

```
--option1 --option2=value2
```

配置文件中不同的选项组是给不同的启动命令使用的，如果选项组名称与程序名称相同，则组中的选项将专门应用于该程序。例如， [mysqld] 和 [mysql] 组分别应用于 mysqld 服务器程序和 mysql 客户端程序。不过有两个选项组比较特别：

- [server] 组下边的启动选项将作用于所有的服务器程序。
- [client] 组下边的启动选项将作用于所有的客户端程序。

需要注意的一点是， mysqld_safe 和 mysql.server 这两个程序在启动时都会读取 [mysqld] 选项组中的内容。为了直观感受一下，我们挑一些启动命令来看一下它们能读取的选项组都有哪些：

启动命令	类别	能读取的组
mysqld	启动服务器	[mysqld] 、 [server]
mysqld_safe	启动服务器	[mysqld] 、 [server] 、 [mysqld_safe]
mysql.server	启动服务器	[mysqld] 、 [server] 、 [mysql.server]
mysql	启动客户端	[mysql] 、 [client]
mysqladmin	启动客户端	[mysqladmin] 、 [client]

启动命令	类别	能读取的组
mysqldump	启动客户端	[mysqldump] 、 [client]

现在我们以 macOS 操作系统为例，在 /etc/mysql/my.cnf 这个配置文件中添加一些内容（ Windows 系统参考上边提到的配置文件路径）：

```
[server]
skip-networking
default-storage-engine=MyISAM
```

然后直接用 mysqld 启动服务器程序：

```
mysqld
```

虽然在命令行没有添加启动选项，但是在程序启动的时候，就会默认的到我们上边提到的配置文件路径下查找配置文件，其中就包括 /etc/mysql/my.cnf 。又由于 mysqld 命令可以读取 [server] 选项组的内容，所以 skip-networking 和 default-storage-engine=MyISAM 这两个选项是生效的。你可以把这些启动选项放在 [client] 组里再试试用 mysqld 启动服务器程序，看一下里边的启动选项生效不（剧透一下，不生效）。

```
小贴士：
如果我们想指定mysql.server程序的启动参数，则必须将它们放在配置文件中，而不是放在命令行中。mysql.server仅支持start和stop作为命令行参数。
```

2.2.3 特定MySQL版本的专用选项组

我们可以在选项组的名称后加上特定的 MySQL 版本号，比如对于 [mysqld] 选项组来说，我们可以定义一个 [mysqld-5.7] 的选项组，它的含义和 [mysqld] 一样，只不过只有版本号为 5.7 的 mysqld 程序才能使用这个选项组中的选项。

2.2.4 配置文件的优先级

我们前边唠叨过 MySQL 将在某些固定的路径下搜索配置文件，我们也可以通过在命令行上指定 defaults-extra-file 启动选项来指定额外的配置文件路径。 MySQL 将按照我们在上表中给定的顺序依次读取各个配置文件，如果该文件不存在则忽略。值得注意的是，如果我们在多个配置文件中设置了相同的启动选项，那以最后一个配置文件中的为准。比方说 /etc/my.cnf 文件的内容是这样的：

```
[server]
default-storage-engine=InnoDB
```

而 ~/.my.cnf 文件中的内容是这样的：

```
[server]
default-storage-engine=MyISAM
```

又因为 ~/.my.cnf 比 /etc/my.cnf 顺序靠后，所以如果两个配置文件中出现相同的启动选项，以 ~/.my.cnf 中的为准，所以 MySQL 服务器程序启动之后， default-storage-engine 的值就是 MyISAM 。

2.2.5 同一个配置文件中多个组的优先级

我们说同一个命令可以访问配置文件中的多个组，比如 mysqld 可以访问 [mysqld] 、 [server] 组，如果在同一个配置文件中，比如 ~/.my.cnf ，在这些组里出现了同样的配置项，比如这样：

```
[server]
default-storage-engine=InnoDB

[mysqld]
default-storage-engine=MyISAM
```

那么，将以最后一个出现的组中的启动选项为准，比方说例子中 default-storage-engine 既出现在 [mysqld] 组也出现在 [server] 组，因为 [mysqld] 组在 [server] 组后边，就以 [mysqld] 组中的配置项为准。

2.2.6 defaults-file的使用

如果我们不想让 MySQL 到默认的路径下搜索配置文件（就是上表中列出的那些），可以在命令行指定 defaults-file 选项，比如这样（以 UNIX 系统为例）：

```
mysqld --defaults-file=/tmp/myconfig.txt
```

这样，在程序启动的时候将只在 /tmp/myconfig.txt 路径下搜索配置文件。如果文件不存在或无法访问，则会发生错误。

```
小贴士：
注意`defaults-extra-file`和`defaults-file`的区别，使用`defaults-extra-file`可以指定额外的
配置文件搜索路径（也就是说那些固定的配置文件路径也会被搜索）。
```

2.3 命令行和配置文件中启动选项的区别

在命令行上指定的绝大部分启动选项都可以放到配置文件中，但是有一些选项是专门为命令行设计的，比方说 defaults-extra-file 、 defaults-file 这样的选项本身就是为了指定配置文件路径的，再放在配置文件中使用就没啥意义了。剩下的一些只能用在命令行上而不能用到配置文件中的启动选项就不一一列举了，用到的时候再提哈（本书中基本用不到，有兴趣的到官方文档看哈）。

另外有一点需要特别注意，如果同一个启动选项既出现在命令行中，又出现在配置文件中，那么以命令行中的启动选项为准！比如我们在配置文件中写了:

```
[server]
default-storage-engine=InnoDB
```

而我们的启动命令是:

```
mysql.server start --default-storage-engine=MyISAM
```

那最后 default-storage-engine 的值就是 MyISAM ！

2.4 系统变量

2.4.1 系统变量简介

MySQL 服务器程序运行过程中会用到许多影响程序行为的变量，它们被称为 MySQL 系统变量，比如允许同时连入的客户端数量用系统变量 max_connections 表示，表的默认存储引擎用系统变量 default_storage_engine 表示，查询缓存的大小用系统变量 query_cache_size 表示， MySQL 服务器程序的系统变量有好几百条，我们就不一一列举了。每个系统变量都有一个默认值，我们可以使用命令行或者配置文件中的选项在启动服务器时改变一些系统变量的值。大多数的系统变量的值也可以在程序运行过程中修改，而无需停止并重新启动它。

2.4.2 查看系统变量

我们可以使用下列命令查看 MySQL 服务器程序支持的系统变量以及它们的当前值:

```
SHOW VARIABLES [LIKE 匹配的模式];
```

由于 系统变量 实在太多了，如果我们直接使用 SHOW VARIABLES 查看的话就直接刷屏了，所以通常都会带一个 LIKE 过滤条件来查看我们需要的系统变量的值，比方说这么写:

```
mysql> SHOW VARIABLES LIKE 'default_storage_engine';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| default_storage_engine | InnoDB |
+------------------------+--------+
1 row in set (0.01 sec)

mysql> SHOW VARIABLES like 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 151   |
+-----------------+-------+
1 row in set (0.00 sec)
```

可以看到，现在服务器程序使用的默认存储引擎就是 InnoDB ，允许同时连接的客户端数量最多为 151 。别忘了 LIKE 表达式后边可以跟通配符来进行模糊查询，也就是说我们可以这么写:

```
mysql> SHOW VARIABLES LIKE 'default%';
+-------------------------------+-----------------------+
| Variable_name                 | Value                 |
+-------------------------------+-----------------------+
| default_authentication_plugin | mysql_native_password |
| default_password_lifetime     | 0                     |
| default_storage_engine        | InnoDB                |
| default_tmp_storage_engine    | InnoDB                |
| default_week_format           | 0                     |
+-------------------------------+-----------------------+
5 rows in set (0.01 sec)

mysql>
```

这样就查出了所有以 default 开头的系统变量的值。

2.4.3 设置系统变量

2.4.3.1 通过启动选项设置

大部分的 系统变量 都可以通过启动服务器时传送启动选项的方式来进行设置。如何填写启动选项我们上边已经花了大篇幅来唠叨了，就是下边两种方式:

- 通过命令行添加启动选项。

  比方说我们在启动服务器程序时用这个命令:

  ```
  mysqld --default-storage-engine=MyISAM --max-connections=10
  ```

- 通过配置文件添加启动选项。

  我们可以这样填写配置文件：

  ```
  [server]
  default-storage-engine=MyISAM
  max-connections=10
  ```

当使用上边两种方式中的任意一种启动服务器程序后，我们再来查看一下系统变量的值：

```
mysql> SHOW VARIABLES LIKE 'default_storage_engine';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| default_storage_engine | MyISAM |
+------------------------+--------+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 10    |
+-----------------+-------+
1 row in set (0.00 sec)

mysql>
```

可以看到 default_storage_engine 和 max_connections 这两个系统变量的值已经被修改了。有一点需要注意的是，对于启动选项来说，如果启动选项名由多个单词组成，各个单词之间用短划线 - 或者下划线 _ 连接起来都可以，但是对应的系统变量之间必须使用下划线 _ 连接起来。

2.4.3.2 服务器程序运行过程中设置

系统变量 比较牛逼的一点就是，对于大部分系统变量来说，它们的值可以在服务器程序运行过程中进行动态修改而无需停止并重启服务器。不过系统变量有作用范围之分，下边详细唠叨下。

设置不同作用范围的系统变量

我们前边说过，多个客户端程序可以同时连接到一个服务器程序。对于同一个系统变量，我们有时想让不同的客户端有不同的值。比方说狗哥使用客户端A，他想让当前客户端对应的默认存储引擎为 InnoDB ，所以他可以把系统变量 default_storage_engine 的值设置为 InnoDB ；猫爷使用客户端B，他想让当前客户端对应的默认存储引擎为 MyISAM ，所以他可以把系统变量 default_storage_engine 的值设置为 MyISAM 。这样可以使狗哥和猫爷的的客户端拥有不同的默认存储引擎，使用时互不影响，十分方便。但是这样各个客户端都私有一份系统变量会产生这么两个问题：

- 有一些系统变量并不是针对单个客户端的，比如允许同时连接到服务器的客户端数量 max_connections ，查询缓存的大小 query_cache_size ，这些公有的系统变量让某个客户端私有显然不合适。
- 一个新连接到服务器的客户端对应的系统变量的值该怎么设置？

为了解决这两个问题，设计 MySQL 的大叔提出了系统变量的 作用范围 的概念，具体来说 作用范围 分为这两种：

- GLOBAL ：全局变量，影响服务器的整体操作。
- SESSION ：会话变量，影响某个客户端连接的操作。 （注： SESSION 有个别名叫 LOCAL ）

在服务器启动时，会将每个全局变量初始化为其默认值（可以通过命令行或选项文件中指定的选项更改这些默认值）。然后服务器还为每个连接的客户端维护一组会话变量，客户端的会话变量在连接时使用相应全局变量的当前值初始化。

这话有点儿绕，还是以 default_storage_engine 举例，在服务器启动时会初始化一个名为 default_storage_engine ，作用范围为 GLOBAL 的系统变量。之后每当有一个客户端连接到该服务器时，服务器都会单独为该客户端分配一个名为 default_storage_engine ，作用范围为 SESSION 的系统变量，该作用范围为 SESSION 的系统变量值按照当前作用范围为 GLOBAL 的同名系统变量值进行初始化。

很显然，通过启动选项设置的系统变量的作用范围都是 GLOBAL 的，也就是对所有客户端都有效的，因为在系统启动的时候还没有客户端程序连接进来呢。了解了系统变量的 GLOBAL 和 SESSION 作用范围之后，我们再看一下在服务器程序运行期间通过客户端程序设置系统变量的语法：

```
SET [GLOBAL|SESSION] 系统变量名 = 值;
```

或者写成这样也行：

```
SET [@@(GLOBAL|SESSION).]var_name = XXX;
```

比如我们想在服务器运行过程中把作用范围为 GLOBAL 的系统变量 default_storage_engine 的值修改为 MyISAM ，也就是想让之后新连接到服务器的客户端都用 MyISAM 作为默认的存储引擎，那我们可以选择下边两条语句中的任意一条来进行设置：

```
语句一：SET GLOBAL default_storage_engine = MyISAM;
语句二：SET @@GLOBAL.default_storage_engine = MyISAM;
```

如果只想对本客户端生效，也可以选择下边三条语句中的任意一条来进行设置：

```
语句一：SET SESSION default_storage_engine = MyISAM;
语句二：SET @@SESSION.default_storage_engine = MyISAM;
语句三：SET default_storage_engine = MyISAM;
```

从上边的 语句三 也可以看出，如果在设置系统变量的语句中省略了作用范围，默认的作用范围就是 SESSION 。也就是说 SET 系统变量名 = 值 和 SET SESSION 系统变量名 = 值 是等价的。

查看不同作用范围的系统变量

既然 系统变量 有 作用范围 之分，那我们的 SHOW VARIABLES 语句查看的是什么 作用范围 的 系统变量 呢？

答：默认查看的是 SESSION 作用范围的系统变量。

当然我们也可以在查看系统变量的语句上加上要查看哪个 作用范围 的系统变量，就像这样：

```
SHOW [GLOBAL|SESSION] VARIABLES [LIKE 匹配的模式];
```

下边我们演示一下完整的设置并查看系统变量的过程：

```
mysql> SHOW SESSION VARIABLES LIKE 'default_storage_engine';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| default_storage_engine | InnoDB |
+------------------------+--------+
1 row in set (0.00 sec)

mysql> SHOW GLOBAL VARIABLES LIKE 'default_storage_engine';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| default_storage_engine | InnoDB |
+------------------------+--------+
1 row in set (0.00 sec)

mysql> SET SESSION default_storage_engine = MyISAM;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW SESSION VARIABLES LIKE 'default_storage_engine';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| default_storage_engine | MyISAM |
+------------------------+--------+
1 row in set (0.00 sec)

mysql> SHOW GLOBAL VARIABLES LIKE 'default_storage_engine';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| default_storage_engine | InnoDB |
+------------------------+--------+
1 row in set (0.00 sec)

mysql>
```

可以看到，最初 `default_storage_engine` 的系统变量无论是在 `GLOBAL` 作用范围上还是在 `SESSION` 作用范围上的值都是 `InnoDB`，我们在 `SESSION` 作用范围把它的值设置为 `MyISAM` 之后，可以看到 `GLOBAL` 作用范围的值并没有改变。

```
小贴士：
如果某个客户端改变了某个系统变量在`GLOBAL`作用范围的值，并不会影响该系统变量在当前已经连接
的客户端作用范围为`SESSION`的值，只会影响后续连入的客户端在作用范围为`SESSION`的值。
```

注意事项

- 并不是所有系统变量都具有 `GLOBAL` 和 `SESSION` 的作用范围。
  - 有一些系统变量只具有 `GLOBAL` 作用范围，比方说 `max_connections`，表示服务器程序支持同时最多有多少个客户端程序进行连接。
  - 有一些系统变量只具有 `SESSION` 作用范围，比如 `insert_id`，表示在对某个包含 `AUTO_INCREMENT` 列的表进行插入时，该列初始的值。

    - 有一些系统变量的值既具有 GLOBAL 作用范围，也具有 SESSION 作用范围，比如我们前边用到的 default_storage_engine ，而且其实大部分的系统变量都是这样的，
- 有些系统变量是只读的，并不能设置值。

  比方说 version ，表示当前 MySQL 的版本，我们客户端是不能设置它的值的，只能在 SHOW VARIABLES 语句里查看。

2.4.4 启动选项和系统变量的区别

启动选项 是在程序启动时我们程序员传递的一些参数，而 系统变量 是影响服务器程序运行行为的变量，它们之间的关系如下：

- 大部分的系统变量都可以被当作启动选项传入。
- 有些系统变量是在程序运行过程中自动生成的，是不可以当作启动选项来设置，比如 auto_increment_offset 、 character_set_client 啥的。
- 有些启动选项也不是系统变量，比如 defaults-file 。

2.5 状态变量

为了让我们更好的了解服务器程序的运行情况， MySQL 服务器程序中维护了好多关于程序运行状态的变量，它们被称为 状态变量 。比方说 Threads_connected 表示当前有多少客户端与服务器建立了连接， Handler_update 表示已经更新了多少行记录吧啦吧啦，像这样显示服务器程序状态信息的 状态变量 还有好几百个，我们就不一一唠叨了，等遇到了会详细说它们的作用的。

由于 状态变量 是用来显示服务器程序运行状况的，所以它们的值只能由服务器程序自己来设置，我们程序员是不能设置的。与 系统变量 类似， 状态变量 也有 GLOBAL 和 SESSION 两个作用范围的，所以查看 状态变量 的语句可以这么写：

```
SHOW [GLOBAL|SESSION] STATUS [LIKE 匹配的模式];
```

类似的，如果我们不写明作用范围，默认的作用范围是 SESSION ，比方说这样：

```
mysql> SHOW STATUS LIKE 'thread%';
+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| Threads_cached    | 0     |
| Threads_connected | 1     |
| Threads_created   | 1     |
| Threads_running   | 1     |
+-------------------+-------+
4 rows in set (0.00 sec)

mysql>
```

所有以 Thread 开头的 SESSION 作用范围的状态变量就都被展示出来了。

3 第3章 乱码的前世今生-字符集和比较规则

标签： MySQL是怎样运行的

3.1 字符集和比较规则简介

3.1.1 字符集简介

我们知道在计算机中只能存储二进制数据，那该怎么存储字符串呢？当然是建立字符与二进制数据的映射关系了，建立这个关系最起码要搞清楚两件事儿：

1. 你要把哪些字符映射成二进制数据？

   也就是界定清楚字符范围。
2. 怎么映射？

   将一个字符映射成一个二进制数据的过程也叫做 `编码` ，将一个二进制数据映射到一个字符的过程叫做 `解码` 。

人们抽象出一个 `字符集` 的概念来描述某个字符范围的编码规则。比方说我们来自定义一个名称为 `xiaohaizi` 的字符集，它包含的字符范围和编码规则如下：

- 包含字符 `'a'` 、 `'b'` 、 `'A'` 、 `'B'` 。
- 编码规则如下：

  采用1个字节编码一个字符的形式，字符和字节的映射关系如下：

  ```
  'a' -> 00000001 (十六进制：0x01)
  'b' -> 00000010 (十六进制：0x02)
  'A' -> 00000011 (十六进制：0x03)
  'B' -> 00000100 (十六进制：0x04)
  ```

有了 `xiaohaizi` 字符集，我们就可以用二进制形式表示一些字符串了，下边是一些字符串用 `xiaohaizi` 字符集编码后的二进制表示：

```
'bA' -> 0000001000000011  (十六进制：0x0203)
'baB' -> 000000100000000100000100  (十六进制：0x020104)
'cd' -> 无法表示，字符集xiaohaizi不包含字符'c'和'd'
```

3.1.2 比较规则简介

在我们确定了 `xiaohaizi` 字符集表示字符的范围以及编码规则后，怎么比较两个字符的大小呢？最容易想到的就是直接比较这两个字符对应的二进制编码的大小，比方说字符 `'a'` 的编码为 `0x01` ，字符 `'b'` 的编码为 `0x02` ，所以 `'a'` 小于 `'b'` ，这种简单的比较规则也可以被称为二进制比较规则，英文名为 `binary collation` 。

二进制比较规则是简单，但有时候并不符合现实需求，比如在很多场合对于英文字符我们都是不区分大小写的，也就是说 `'a'` 和 `'A'` 是相等的，在这种场合下就不能简单粗暴的使用二进制比较规则了，这时候我们可以这样指定比较规则：

1. 将两个大小写不同的字符全都转为大写或者小写。
2. 再比较这两个字符对应的二进制数据。

这是一种稍微复杂一点点的比较规则，但是实际生活中的字符不止英文字符一种，比如我们的汉字有几万之多，对于某一种字符集来说，比较两个字符大小的规则可以制定出很多种，也就是说同一种字符集可以有多种比较规则，我们稍后就要介绍各种现实生活中用的字符集以及它们的一些比较规则。

3.1.3 一些重要的字符集

不幸的是，这个世界太大了，不同的人制定出了好多种 字符集 ，它们表示的字符范围和用到的编码规则可能都不一样。我们看一下一些常用字符集的情况：

- ASCII 字符集

  共收录128个字符，包括空格、标点符号、数字、大小写字母和一些不可见字符。由于总共才128个字符，所以可以使用1个字节来进行编码，我们看一些字符的编码方式：

  ```
  'L' ->  01001100（十六进制：0x4C，十进制：76）
  'M' ->  01001101（十六进制：0x4D，十进制：77）
  ```

- ISO 8859-1 字符集

  共收录256个字符，是在 ASCII 字符集的基础上又扩充了128个西欧常用字符(包括德法两国的字母)，也可以使用1个字节来进行编码。这个字符集也有一个别名 latin1 。

- GB2312 字符集

  收录了汉字以及拉丁字母、希腊字母、日文平假名及片假名字母、俄语西里尔字母。其中收录汉字6763个，其他文字符号682个。同时这种字符集又兼容 ASCII 字符集，所以在编码方式上显得有些奇怪：

  - 如果该字符在 ASCII 字符集中，则采用1字节编码。
  - 否则采用2字节编码。

    这种表示一个字符需要的字节数可能不同的编码方式称为 变长编码方式 。比方说字符串 '爱u' ，其中 '爱' 需要用2个字节进行编码，编码后的十六进制表示为 0xCED2 ， 'u' 需要用1个字节进行编码，编码后的十六进制表示为 0x75 ，所以拼合起来就是 0xCED275 。

    > 小贴士：
    > 我们怎么区分某个字节代表一个单独的字符还是代表某个字符的一部分呢？别忘了`ASCII`字符集只收录128个字符，使用0～127就可以表示全部字符，所以如果某个字节是在0～127之内的，就意味着一个字节代表一个单独的字符，否则就是两个字节代表一个单独的字符。

- GBK 字符集

  GBK 字符集只是在收录字符范围上对 GB2312 字符集作了扩充，编码方式上兼容 GB2312 。

- utf8 字符集

  收录地球上能想到的所有字符，而且还在不断扩充。这种字符集兼容 ASCII 字符集，采用变长编码方式，编码一个字符需要使用1～4个字节，比方说这样：

  ```
  'L' ->  01001100（十六进制：0x4C）
  '啊' ->  111001011001010110001010（十六进制：0xE5958A）
  ```

  > 小贴士：
  > 其实准确的说，utf8只是Unicode字符集的一种编码方案，Unicode字符集可以采用utf8、utf16、utf32这几种编码方案，utf8使用1～4个字节编码一个字符，utf16使用2个或4个字节编码一个字符，utf32使用4个字节编码一个字符。更详细的Unicode和其编码方案的知识不是本书的重点，大家上网查查哈～
  >
  > MySQL中并不区分字符集和编码方案的概念，所以后边唠叨的时候把utf8、utf16、utf32都当作一种字符集对待。

对于同一个字符，不同字符集也可能有不同的编码方式。比如对于汉字 '我' 来说， ASCII 字符集中根本没有收录这个字符， utf8 和 gb2312 字符集对汉字 我 的编码方式如下：

```
utf8编码：111001101000100010010001 (3个字节，十六进制表示是：0xE68891)
gb2312编码：1100111011010010 (2个字节，十六进制表示是：0xCED2)
```

3.2 MySQL中支持的字符集和排序规则

3.2.1 MySQL中的utf8和utf8mb4

我们上边说 `utf8` 字符集表示一个字符需要使用1～4个字节，但是我们常用的一些字符使用1～3个字节就可以表示了。而在 `MySQL` 中字符集表示一个字符所用最大字节长度在某些方面会影响系统的存储和性能，所以设计 `MySQL` 的大叔偷偷的定义了两个概念：

- `utf8mb3` ：阉割过的 `utf8` 字符集，只使用1～3个字节表示字符。
- `utf8mb4` ：正宗的 `utf8` 字符集，使用1～4个字节表示字符。

有一点需要大家十分的注意，在 `MySQL` 中 `utf8` 是 `utf8mb3` 的别名，所以之后在 `MySQL` 中提到 `utf8` 就意味着使用1~3个字节来表示一个字符，如果大家有使用4字节编码一个字符的情况，比如存储一些emoji表情啥的，那请使用 `utf8mb4` 。

3.2.2 字符集的查看

`MySQL` 支持好多好多种字符集，查看当前 `MySQL` 中支持的字符集可以用下边这个语句：

```
SHOW (CHARACTER SET|CHARSET) [LIKE 匹配的模式];
```

其中 `CHARACTER SET` 和 `CHARSET` 是同义词，用任意一个都可以。我们查询一下（支持的字符集太多了，我们省略了一些）：

```
mysql> SHOW CHARSET;
+----------+---------------------------------+---------------------+--------+
| Charset  | Description                     | Default collation   | Maxlen |
+----------+---------------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese        | big5_chinese_ci     |      2 |
...
| latin1   | cp1252 West European            | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European     | latin2_general_ci   |      1 |
...
| ascii    | US ASCII                        | ascii_general_ci    |      1 |
...
| gb2312   | GB2312 Simplified Chinese       | gb2312_chinese_ci   |      2 |
...
| gbk      | GBK Simplified Chinese          | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish              | latin5_turkish_ci   |      1 |
...
| utf8     | UTF-8 Unicode                   | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode                   | ucs2_general_ci     |      2 |
...
| latin7   | ISO 8859-13 Baltic              | latin7_general_ci   |      1 |
| utf8mb4  | UTF-8 Unicode                   | utf8mb4_general_ci  |      4 |
| utf16    | UTF-16 Unicode                  | utf16_general_ci    |      4 |
| utf16le  | UTF-16LE Unicode                | utf16le_general_ci  |      4 |
...
| utf32    | UTF-32 Unicode                  | utf32_general_ci    |      4 |
| binary   | Binary pseudo charset           | binary              |      1 |
...
| gb18030  | China National Standard GB18030 | gb18030_chinese_ci  |      4 |
+----------+---------------------------------+---------------------+--------+
41 rows in set (0.01 sec)
```

可以看到，我使用的这个 MySQL 版本一共支持 41 种字符集，其中的 Default collation 列表示这种字符集中一种默认的 比较规则 。大家注意返回结果中的最后一列 Maxlen ，它代表该种字符集表示一个字符最多需要几个字节。为了让大家的印象更深刻，我把几个常用到的字符集的 Maxlen 列摘抄下来，大家务必记住：

字符集名称	Maxlen
ascii	1
latin1	1
gb2312	2
gbk	2
utf8	3
utf8mb4	4

3.2.3 比较规则的查看

查看 MySQL 中支持的比较规则的命令如下：

```
SHOW COLLATION [LIKE 匹配的模式];
```

我们前边说过一种字符集可能对应着若干种比较规则， MySQL 支持的字符集就已经非常多了，所以支持的比较规则更多，我们先只查看一下 utf8 字符集下的比较规则：

```
mysql> SHOW COLLATION LIKE 'utf8\_%';
+--------------------------+---------+-----+---------+----------+---------+
| Collation                | Charset | Id  | Default | Compiled | Sortlen |
+--------------------------+---------+-----+---------+----------+---------+
| utf8_general_ci          | utf8    |  33 | Yes     | Yes      |       1 |
| utf8_bin                 | utf8    |  83 |         | Yes      |       1 |
| utf8_unicode_ci          | utf8    | 192 |         | Yes      |       8 |
| utf8_icelandic_ci        | utf8    | 193 |         | Yes      |       8 |
| utf8_latvian_ci          | utf8    | 194 |         | Yes      |       8 |
| utf8_romanian_ci         | utf8    | 195 |         | Yes      |       8 |
| utf8_slovenian_ci        | utf8    | 196 |         | Yes      |       8 |
| utf8_polish_ci           | utf8    | 197 |         | Yes      |       8 |
| utf8_estonian_ci         | utf8    | 198 |         | Yes      |       8 |
| utf8_spanish_ci          | utf8    | 199 |         | Yes      |       8 |
| utf8_swedish_ci          | utf8    | 200 |         | Yes      |       8 |
| utf8_turkish_ci          | utf8    | 201 |         | Yes      |       8 |
| utf8_czech_ci            | utf8    | 202 |         | Yes      |       8 |
| utf8_danish_ci           | utf8    | 203 |         | Yes      |       8 |
| utf8_lithuanian_ci       | utf8    | 204 |         | Yes      |       8 |
| utf8_slovak_ci           | utf8    | 205 |         | Yes      |       8 |
| utf8_spanish2_ci         | utf8    | 206 |         | Yes      |       8 |
| utf8_roman_ci            | utf8    | 207 |         | Yes      |       8 |
| utf8_persian_ci          | utf8    | 208 |         | Yes      |       8 |
| utf8_esperanto_ci        | utf8    | 209 |         | Yes      |       8 |
| utf8_hungarian_ci        | utf8    | 210 |         | Yes      |       8 |
| utf8_sinhala_ci          | utf8    | 211 |         | Yes      |       8 |
| utf8_german2_ci          | utf8    | 212 |         | Yes      |       8 |
| utf8_croatian_ci         | utf8    | 213 |         | Yes      |       8 |
| utf8_unicode_520_ci      | utf8    | 214 |         | Yes      |       8 |
| utf8_vietnamese_ci       | utf8    | 215 |         | Yes      |       8 |
| utf8_general_mysql500_ci | utf8    | 223 |         | Yes      |       1 |
+--------------------------+---------+-----+---------+----------+---------+
27 rows in set (0.00 sec)
```

这些比较规则的命名还挺有规律的，具体规律如下：

- 比较规则名称以与其关联的字符集的名称开头。如上图的查询结果的比较规则名称都是以 utf8 开头的。
- 后边紧跟着该比较规则主要作用于哪种语言，比如 utf8_polish_ci 表示以波兰语的规则比较， utf8_spanish_ci 是以西班牙语的规则比较， utf8_general_ci 是一种通用的比较规则。
- 名称后缀意味着该比较规则是否区分语言中的重音、大小写啥的，具体可以用的值如下：

  |后缀|英文释义|描述| |:--:|:--:|:--:| | _ai | accent insensitive |不区分重音| | _as | accent sensitive |区分重音| | _ci | case insensitive |不区分大小写| | _cs | case sensitive |区分大小写| | _bin | binary |以二进制方式比较|

  比如 utf8_general_ci 这个比较规则是以 ci 结尾的，说明不区分大小写。

每种字符集对应若干种比较规则，每种字符集都有一种默认的比较规则， SHOW COLLATION 的返回结果中的 Default 列的值为 YES 的就是该字符集的默认比较规则，比方说 utf8 字符集默认的比较规则就是 utf8_general_ci 。

3.3 字符集和比较规则的应用

3.3.1 各级别的字符集和比较规则

MySQL 有4个级别的字符集和比较规则，分别是：

- 服务器级别
- 数据库级别
- 表级别
- 列级别

我们接下来仔细看一下怎么设置和查看这几个级别的字符集和比较规则。

3.3.1.1 服务器级别

MySQL 提供了两个系统变量来表示服务器级别的字符集和比较规则：

系统变量	描述
character_set_server	服务器级别的字符集
collation_server	服务器级别的比较规则

我们看一下这两个系统变量的值：

```
mysql> SHOW VARIABLES LIKE 'character_set_server';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| character_set_server | utf8  |
+----------------------+-------+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'collation_server';
+------------------+-----------------+
| Variable_name    | Value           |
+------------------+-----------------+
| collation_server | utf8_general_ci |
+------------------+-----------------+
1 row in set (0.00 sec)
```

可以看到在我的计算机中服务器级别默认的字符集是 utf8 ，默认的比较规则是 utf8_general_ci 。

我们可以在启动服务器程序时通过启动选项或者在服务器程序运行过程中使用 SET 语句修改这两个变量的值。比如我们可以在配置文件中这样写：

```
[server]
character_set_server=gbk
collation_server=gbk_chinese_ci
```

当服务器启动的时候读取这个配置文件后这两个系统变量的值便修改了。

3.3.1.2 数据库级别

我们在创建和修改数据库的时候可以指定该数据库的字符集和比较规则，具体语法如下：

```
CREATE DATABASE 数据库名
    [[DEFAULT] CHARACTER SET 字符集名称]
    [[DEFAULT] COLLATE 比较规则名称];

ALTER DATABASE 数据库名
    [[DEFAULT] CHARACTER SET 字符集名称]
    [[DEFAULT] COLLATE 比较规则名称];
```

其中的 DEFAULT 可以省略，并不影响语句的语义。比方说我们新创建一个名叫 charset_demo_db 的数据库，在创建的时候指定它使用的字符集为 gb2312 ，比较规则为 gb2312_chinese_ci ：

```
mysql> CREATE DATABASE charset_demo_db
    -> CHARACTER SET gb2312
    -> COLLATE gb2312_chinese_ci;
Query OK, 1 row affected (0.01 sec)
```

如果想查看当前数据库使用的字符集和比较规则，可以查看下面两个系统变量的值（前提是使用 USE 语句选择当前默认数据库，如果没有默认数据库，则变量与相应的服务器级系统变量具有相同的值）：

系统变量	描述
character_set_database	当前数据库的字符集
collation_database	当前数据库的比较规则

我们来查看一下刚刚创建的 charset_demo_db 数据库的字符集和比较规则：

```
mysql> USE charset_demo_db;
Database changed

mysql> SHOW VARIABLES LIKE 'character_set_database';
+------------------------+--------+
| Variable_name          | Value  |
+------------------------+--------+
| character_set_database | gb2312 |
+------------------------+--------+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'collation_database';
+--------------------+-------------------+
| Variable_name      | Value             |
+--------------------+-------------------+
| collation_database | gb2312_chinese_ci |
+--------------------+-------------------+
1 row in set (0.00 sec)

mysql>
```

可以看到这个 charset_demo_db 数据库的字符集和比较规则就是我们在创建语句中指定的。需要注意的一点是：***character_set_database* 和 *collation_database* 这两个系统变量是只读的，我们不能通过修改这两个变量的值而改变当前数据库的字符集和比较规则**。

数据库的创建语句中也可以不指定字符集和比较规则，比如这样：

```
CREATE DATABASE 数据库名;
```

这样的话将使用服务器级别的字符集和比较规则作为数据库的字符集和比较规则。

3.3.1.3 表级别

我们也可以在创建和修改表的时候指定表的字符集和比较规则，语法如下：

```
CREATE TABLE 表名 (列的信息)
    [[DEFAULT] CHARACTER SET 字符集名称]
    [COLLATE 比较规则名称]]

ALTER TABLE 表名
    [[DEFAULT] CHARACTER SET 字符集名称]
    [COLLATE 比较规则名称]
```

比方说我们在刚刚创建的 charset_demo_db 数据库中创建一个名为 t 的表，并指定这个表的字符集和比较规则：

```
mysql> CREATE TABLE t(
    ->     col VARCHAR(10)
    -> ) CHARACTER SET utf8 COLLATE utf8_general_ci;
Query OK, 0 rows affected (0.03 sec)
```

如果创建和修改表的语句中没有指明字符集和比较规则，将使用该表所在数据库的字符集和比较规则作为该表的字符集和比较规则。假设我们的创建表 t 的语句是这么写的：

```
CREATE TABLE t(
    col VARCHAR(10)
);
```

因为表 t 的建表语句中并没有明确指定字符集和比较规则，则表 t 的字符集和比较规则将继承所在数据库 charset_demo_db 的字符集和比较规则，也就是 gb2312 和 gb2312_chinese_ci 。

3.3.1.4 列级别

需要注意的是，对于存储字符串的列，同一个表中的不同的列也可以有不同的字符集和比较规则。我们在创建和修改列定义的时候可以指定该列的字符集和比较规则，语法如下：

```
CREATE TABLE 表名(
    列名 字符串类型 [CHARACTER SET 字符集名称] [COLLATE 比较规则名称],
    其他列...
);

ALTER TABLE 表名 MODIFY 列名 字符串类型 [CHARACTER SET 字符集名称] [COLLATE 比较规则名称];
```

比如我们修改一下表 t 中列 col 的字符集和比较规则可以这么写：

```
mysql> ALTER TABLE t MODIFY col VARCHAR(10) CHARACTER SET gbk COLLATE gbk_chinese_ci;
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
```

对于某个列来说，如果在创建和修改的语句中没有指明字符集和比较规则，将使用该列所在表的字符集和比较规则作为该列的字符集和比较规则。比方说表 t 的字符集是 utf8 ，比较规则是 utf8_general_ci ，修改列 col 的语句是这么写的：

```
ALTER TABLE t MODIFY col VARCHAR(10);
```

```
ALTER TABLE t MODIFY col VARCHAR(10);
```

那列 col 的字符集和编码将使用表 t 的字符集和比较规则，也就是 utf8 和 utf8_general_ci 。

> 小贴士：
> 在转换列的字符集时需要注意，如果转换前列中存储的数据不能用转换后的字符集进行表示会发生错误。比方说原先列使用的字符集是utf8，列中存储了一些汉字，现在把列的字符集转换为ascii的话就会出错，因为ascii字符集并不能表示汉字字符。

3.3.1.5 仅修改字符集或仅修改比较规则

由于字符集和比较规则是互相有联系的，如果我们只修改了字符集，比较规则也会跟着变化，如果只修改了比较规则，字符集也会跟着变化，具体规则如下：

- 只修改字符集，则比较规则将变为修改后的字符集默认的比较规则。
- 只修改比较规则，则字符集将变为修改后的比较规则对应的字符集。

不论哪个级别的字符集和比较规则，这两条规则都适用，我们以服务器级别的字符集和比较规则为例来看一下详细过程：

- 只修改字符集，则比较规则将变为修改后的字符集默认的比较规则。

```
mysql> SET character_set_server = gb2312;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'character_set_server';
+----------------------+--------+
| Variable_name        | Value  |
+----------------------+--------+
| character_set_server | gb2312 |
+----------------------+--------+
1 row in set (0.00 sec)

mysql>  SHOW VARIABLES LIKE 'collation_server';
+------------------+-------------------+
| Variable_name    | Value             |
+------------------+-------------------+
| collation_server | gb2312_chinese_ci |
+------------------+-------------------+
1 row in set (0.00 sec)
```

  我们只修改了 character_set_server 的值为 gb2312 ， collation_server 的值自动变为了 gb2312_chinese_ci 。

- 只修改比较规则，则字符集将变为修改后的比较规则对应的字符集。

```
mysql> SET collation_server = utf8_general_ci;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'character_set_server';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| character_set_server | utf8  |
+----------------------+-------+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'collation_server';
+------------------+-----------------+
| Variable_name    | Value           |
+------------------+-----------------+
| collation_server | utf8_general_ci |
+------------------+-----------------+
1 row in set (0.00 sec)

mysql>
```

我们只修改了 `collation_server` 的值为 `utf8_general_ci` ， `character_set_server` 的值自动变为了 `utf8` 。

3.3.1.6 各级别字符集和比较规则小结

我们介绍的这4个级别字符集和比较规则的联系如下：

- 如果创建或修改列时没有显式的指定字符集和比较规则，则该列默认用表的字符集和比较规则
- 如果创建或修改表时没有显式的指定字符集和比较规则，则该表默认用数据库的字符集和比较规则
- 如果创建或修改数据库时没有显式的指定字符集和比较规则，则该数据库默认用服务器的字符集和比较规则

知道了这些规则之后，对于给定的表，我们应该知道它的各个列的字符集和比较规则是什么，从而根据这个列的类型来确定存储数据时每个列的实际数据占用的存储空间大小了。比方说我们向表 `t` 中插入一条记录：

```
mysql> INSERT INTO t(col) VALUES('我我');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t;
+--------+
| s      |
+--------+
| 我我   |
+--------+
1 row in set (0.00 sec)
```

首先列 `col` 使用的字符集是 `gbk` ，一个字符 `'我'` 在 `gbk` 中的编码为 `0xCED2` ，占用两个字节，两个字符的实际数据就占用4个字节。如果把该列的字符集修改为 `utf8` 的话，这两个字符就实际占用6个字节啦~

3.3.2 客户端和服务器通信中的字符集

3.3.2.1 编码和解码使用的字符集不一致的后果

说到底，字符串在计算机上的体现就是一个字节串，如果你使用不同字符集去解码这个字节串，最后得到的结果可能让你挠头。

我们知道字符 '我' 在 utf8 字符集编码下的字节串长这样： 0xE68891 ，如果一个程序把这个字节串发送到另一个程序里，另一个程序用不同的字符集去解码这个字节串，假设使用的是 gbk 字符集来解释这串字节，解码过程就是这样的：

1. 首先看第一个字节 0xE6 ，它的值大于 0x7F （十进制：127），说明是两字节编码，继续读一字节后是 0xE688 ，然后从 gbk 编码表中查找字节为 0xE688 对应的字符，发现是字符 '鎴'
2. 继续读一个字节 0x91 ，它的值也大于 0x7F ，再往后读一个字节发现木有了，所以这是半个字符。
3. 所以 0xE68891 被 gbk 字符集解释成一个字符 '鎴' 和半个字符。

假设用 iso-8859-1 ，也就是 latin1 字符集去解释这串字节，解码过程如下：

1. 先读第一个字节 0xE6 ，它对应的 latin1 字符为 æ 。
2. 再读第二个字节 0x88 ，它对应的 latin1 字符为 ˆ 。
3. 再读第二个字节 0x91 ，它对应的 latin1 字符为 ‘ 。
4. 所以整串字节 0xE68891 被 latin1 字符集解释后的字符串就是 'æˆ‘'

可见，如果对于同一个字符串编码和解码使用的字符集不一样，会产生意想不到的结果，作为人类的我们看上去就像是产生了乱码一样。

3.3.2.2 字符集转换的概念

如果接收 0xE68891 这个字节串的程序按照 utf8 字符集进行解码，然后又把它按照 gbk 字符集进行编码，最后编码后的字节串就是 0xCED2 ，我们把这个过程称为 字符集的转换 ，也就是字符串 '我' 从 utf8 字符集转换为 gbk 字符集。

3.3.2.3 MySQL中字符集的转换

我们知道从客户端发往服务器的请求本质上就是一个字符串，服务器向客户端返回的结果本质上也是一个字符串，而字符串其实是使用某种字符集编码的二进制数据。这个字符串可不是使用一种字符集的编码方式一条道走到黑的，从发送请求到返回结果这个过程中伴随着多次字符集的转换，在这个过程中会用到3个系统变量，我们先把它们写出来看一下：

系统变量	描述
character_set_client	服务器解码请求时使用的字符集
character_set_connection	服务器处理请求时会把请求字符串从 character_set_client 转为 character_set_connection
character_set_results	服务器向客户端返回数据时使用的字符集

这几个系统变量在我的计算机上的默认值如下（不同操作系统的默认值可能不同）：

```
mysql> SHOW VARIABLES LIKE 'character_set_client';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| character_set_client | utf8  |
+----------------------+-------+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'character_set_connection';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| character_set_connection | utf8  |
+--------------------------+-------+
1 row in set (0.01 sec)

mysql> SHOW VARIABLES LIKE 'character_set_results';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| character_set_results | utf8  |
+-----------------------+-------+
1 row in set (0.00 sec)
```

大家可以看到这几个系统变量的值都是 `utf8` ，为了体现出字符集在请求处理过程中的变化，我们这里特意修改一个系统变量的值：

```
mysql> set character_set_connection = gbk;
Query OK, 0 rows affected (0.00 sec)
```

所以现在系统变量 `character_set_client` 和 `character_set_results` 的值还是 `utf8` ，而 `character_set_connection` 的值为 `gbk` 。现在假设我们客户端发送的请求是下边这个字符串：

```
SELECT * FROM t WHERE s = '我';
```

为了方便大家理解这个过程，我们只分析字符 `'我'` 在这个过程中字符集的转换。

现在看一下在请求从发送到结果返回过程中字符集的变化：

1. 客户端发送请求所使用的字符集

   一般情况下客户端所使用的字符集和当前操作系统一致，不同操作系统使用的字符集可能不一样，如下：

   - 类 `Unix` 系统使用的是 `utf8`
   - `Windows` 使用的是 `gbk`

     例如我在使用的 `macOS` 操作系统时，客户端使用的就是 `utf8` 字符集。所以字符 `'我'` 在发送给服务器的请求中的字节形式就是： `0xE68891`

     > 小贴士：
     > 如果你使用的是可视化工具，比如navicat之类的，这些工具可能会使用自定义的字符集来编码发送到服务器的字符串，而不采用操作系统默认的字符集（所以在学习的时候还是尽量用黑框框哈）。

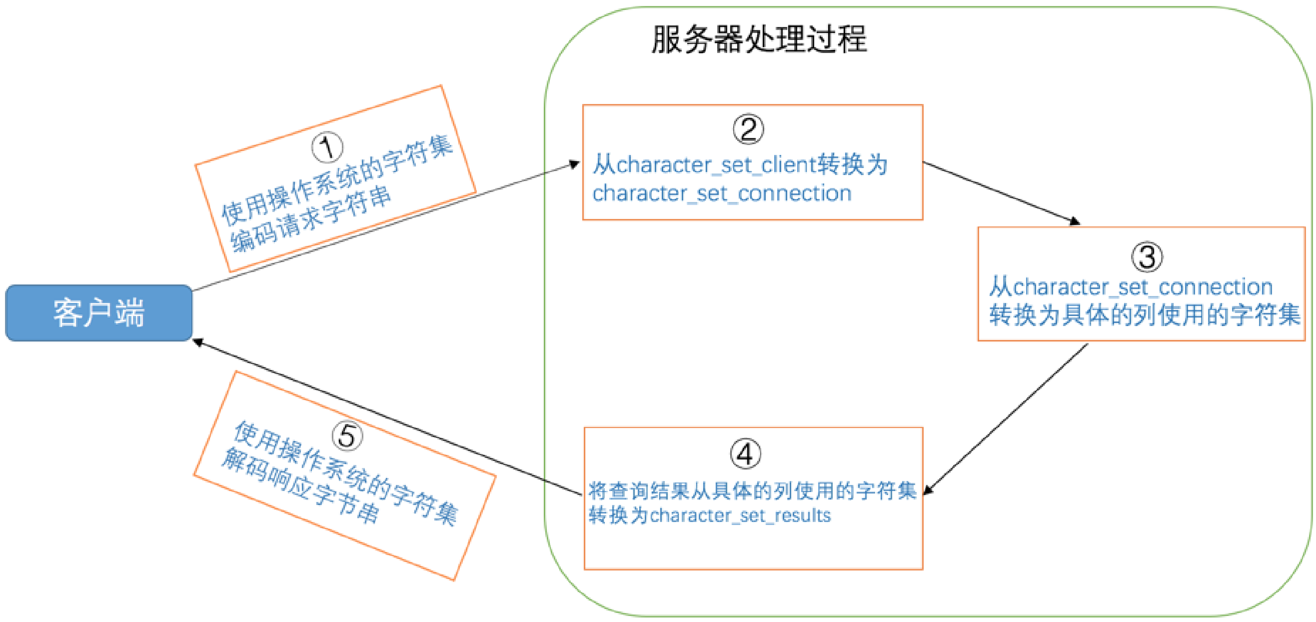
2. 服务器接收到客户端发送来的请求其实是一串二进制的字节，它会认为这串字节采用的字符集是 character_set_client ，然后把这串字节转换为 character_set_connection 字符集编码的字符。

   由于我的计算机上 character_set_client 的值是 utf8 ，首先会按照 utf8 字符集对字节串 0xE68891 进行解码，得到的字符串就是 '我' ，然后按照 character_set_connection 代表的字符集，也就是 gbk 进行编码，得到的结果就是字节串 0xCED2 。
3. 因为表 t 的列 col 采用的是 gbk 字符集，与 character_set_connection 一致，所以直接到列中找字节值为 0xCED2 的记录，最后找到了一条记录。

   > 小贴士：
   > 如果某个列使用的字符集和character_set_connection代表的字符集不一致的话，还需要进行一次字符集转换。

4. 上一步骤找到的记录中的 col 列其实是一个字节串 0xCED2 ， col 列是采用 gbk 进行编码的，所以首先会将这个字节串使用 gbk 进行解码，得到字符串 '我' ，然后再把这个字符串使用 character_set_results 代表的字符集，也就是 utf8 进行编码，得到了新的字节串： 0xE68891 ，然后发送给客户端。
5. 由于客户端是用的字符集是 utf8 ，所以可以顺利的将 0xE68891 解释成字符 我 ，从而显示到我们的显示器上，所以我们人类也读懂了返回的结果。

如果你读上边的文字有点晕，可以参照这个图来仔细分析一下这几个步骤：



从这个分析中我们可以得出这么几点需要注意的地方：

- 服务器认为客户端发送过来的请求是用 character_set_client 编码的。

  假设你的客户端采用的字符集和 *character_set_client* 不一样的话，这就会出现意想不到的情况。比如我的客户端使用的是 utf8 字符集，如果把系统变量 character_set_client 的值设置为 ascii 的话，服务器可能无法理解我们发送的请求，更别谈处理这个请求了。
- 服务器将把得到的结果集使用 character_set_results 编码后发送给客户端。

  假设你的客户端采用的字符集和 *character_set_results* 不一样的话，这就可能会出现客户端无法解码结果集的情况，结果就是在你的屏幕上出现乱码。比如我的客户端使用的是 utf8 字符集，如果把系统变量 character_set_results 的值设置为 ascii 的话，可能会产生乱码。
- character_set_connection 只是服务器在将请求的字节串从 character_set_client 转换为 character_set_connection 时使用，它是什么其实没多重要，但是一定要注意，**该字符集包含的字符范围一定涵盖请求中的字符**，要不然会导致有的字符无法使用 character_set_connection 代表的字符集进行编

码。比如你把 character_set_client 设置为 utf8 ，把 character_set_connection 设置成 ascii ，那么此时你如果从客户端发送一个汉字到服务器，那么服务器无法使用 ascii 字符集来编码这个汉字，就会向用户发出一个警告。

知道了在 MySQL 中从发送请求到返回结果过程里发生的各种字符集转换，但是为啥要转来转去的呢？不晕么？

答：是的，很头晕，所以我们通常都把 ***character_set_client*** 、***character_set_connection***、***character_set_results*** 这三个系统变量设置成和客户端使用的字符集一致的情况，这样减少了很多无谓的字符集转换。为了方便我们设置， MySQL 提供了一条非常简便的语句：

```
SET NAMES 字符集名;
```

这一条语句产生的效果和我们执行这3条的效果是一样的：

```
SET character_set_client = 字符集名;
SET character_set_connection = 字符集名;
SET character_set_results = 字符集名;
```

比方说我的客户端使用的是 utf8 字符集，所以需要把这几个系统变量的值都设置为 utf8 ：

```
mysql> SET NAMES utf8;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'character_set_client';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| character_set_client | utf8  |
+----------------------+-------+
1 row in set (0.00 sec)

mysql>  SHOW VARIABLES LIKE 'character_set_connection';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| character_set_connection | utf8  |
+--------------------------+-------+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'character_set_results';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| character_set_results | utf8  |
+-----------------------+-------+
1 row in set (0.00 sec)

mysql>
```

```
小贴士：
如果你使用的是Windows系统，那应该设置成gbk。
```

另外，如果你想在启动客户端的时候就把 character_set_client 、 character_set_connection 、 character_set_results 这三个系统变量的值设置成一样的，那我们可以在启动客户端的时候指定一个叫 default-character-set 的启动选项，比如在配置文件里可以这么写：

default-character-set 的启动选项，比如在配置文件里可以这么写：

```
[client]
default-character-set=utf8
```

它起到的效果和执行一遍 SET NAMES utf8 是一样一样的，都会将那三个系统变量的值设置成 utf8 。

3.3.3 比较规则的应用

结束了字符集的漫游，我们把视角再次聚焦到 比较规则 ， 比较规则 的作用通常体现比较字符串大小的表达式以及对某个字符串列进行排序中，所以有时候也称为 排序规则 。比方说表 t 的列 col 使用的字符集是 gbk ，使用的比较规则是 gbk_chinese_ci ，我们向里边插入几条记录：

```
mysql> INSERT INTO t(col) VALUES('a'), ('b'), ('A'), ('B');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql>
```

我们查询的时候按照 t 列排序一下：

```
mysql> SELECT * FROM t ORDER BY col;
+------+
| col  |
+------+
| a    |
| A    |
| b    |
| B    |
| 我   |
+------+
5 rows in set (0.00 sec)
```

可以看到在默认的比较规则 gbk_chinese_ci 中是不区分大小写的，我们现在把列 col 的比较规则修改为 gbk_bin ：

```
mysql> ALTER TABLE t MODIFY col VARCHAR(10) COLLATE gbk_bin;
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

由于 gbk_bin 是直接比较字符的编码，所以是区分大小写的，我们再看一下排序后的查询结果：

```
mysql> SELECT * FROM t ORDER BY s;
+------+
| s    |
+------+
| A    |
| B    |
| a    |
| b    |
| 我   |
+------+
5 rows in set (0.00 sec)

mysql>
```

所以如果以后大家在对字符串做比较或者对某个字符串列做排序操作时没有得到想象中的结果，需要思考一下是不是 比较规则 的问题~

> 小贴士：
> 列`col`中各个字符在使用gbk字符集编码后对应的数字如下：
> 'A' -> 65 （十进制）
> 'B' -> 66 （十进制）
> 'a' -> 97 （十进制）
> 'b' -> 98 （十进制）
> '我' -> 25105 （十进制）

3.4 总结

1. 字符集 指的是某个字符范围的编码规则。
2. 比较规则 是针对某个字符集中的字符比较大小的一种规则。
3. 在 MySQL 中，一个字符集可以有若干种比较规则，其中有一个默认的比较规则，一个比较规则必须对应一个字符集。
4. 查看 MySQL 中查看支持的字符集和比较规则的语句如下：

```
SHOW (CHARACTER SET|CHARSET) [LIKE 匹配的模式];
SHOW COLLATION [LIKE 匹配的模式];
```

5. MySQL有四个级别的字符集和比较规则

- 服务器级别

  character_set_server 表示服务器级别的字符集， collation_server 表示服务器级别的比较规则。

- 数据库级别

  创建和修改数据库时可以指定字符集和比较规则：

```
CREATE DATABASE 数据库名
    [[DEFAULT] CHARACTER SET 字符集名称]
    [[DEFAULT] COLLATE 比较规则名称];

ALTER DATABASE 数据库名
    [[DEFAULT] CHARACTER SET 字符集名称]
    [[DEFAULT] COLLATE 比较规则名称];
```

  character_set_database 表示当前数据库的字符集， collation_database 表示当前默认数据库的比较规则，这两个系统变量是只读的，不能修改。如果没有指定当前默认数据库，则变量与相应的服务器级系统变量具有相同的值。

- 表级别

  创建和修改表的时候指定表的字符集和比较规则：

```
CREATE TABLE 表名 (列的信息)
    [[DEFAULT] CHARACTER SET 字符集名称]
    [COLLATE 比较规则名称]];

ALTER TABLE 表名
    [[DEFAULT] CHARACTER SET 字符集名称]
    [COLLATE 比较规则名称];
```

- 列级别

创建和修改列定义的时候可以指定该列的字符集和比较规则：

```
CREATE TABLE 表名(
    列名 字符串类型 [CHARACTER SET 字符集名称] [COLLATE 比较规则名称],
    其他列...
);

ALTER TABLE 表名 MODIFY 列名 字符串类型 [CHARACTER SET 字符集名称] [COLLATE 比较规
则名称];
```

6. 从发送请求到接收结果过程中发生的字符集转换：

- 客户端使用操作系统的字符集编码请求字符串，向服务器发送的是经过编码的一个字节串。
- 服务器将客户端发送来的字节串采用 `character_set_client` 代表的字符集进行解码，将解码后的字符串再按照 `character_set_connection` 代表的字符集进行编码。
- 如果 `character_set_connection` 代表的字符集和具体操作的列使用的字符集一致，则直接进行相应操作，否则的话需要将请求中的字符串从 `character_set_connection` 代表的字符集转换为具体操作的列使用的字符集之后再进行操作。
- 将从某个列获取到的字节串从该列使用的字符集转换为 `character_set_results` 代表的字符集后发送到客户端。
- 客户端使用操作系统的字符集解析收到的结果集字节串。

  在这个过程中各个系统变量的含义如下：

  |系统变量|描述| |:--:|:--:| | `character_set_client` |服务器解码请求时使用的字符集| | `character_set_connection` |服务器处理请求时会把请求字符串从 `character_set_client` 转为 `character_set_connection` | | `character_set_results` |服务器向客户端返回数据时使用的字符集|

  一般情况下要使用保持这三个变量的值和客户端使用的字符集相同。

7. 比较规则的作用通常体现比较字符串大小的表达式以及对某个字符串列进行排序中。

4 第4章 从一条记录说起-InnoDB记录结构

标签： MySQL是怎样运行的

4.1 准备工作

到现在为止， `MySQL` 对于我们来说还是一个黑盒，我们只负责使用客户端发送请求并等待服务器返回结果，表中的数据到底存到了哪里？以什么格式存放的？ `MySQL` 是以什么方式来访问的这些数据？这些问题我们统统不知道，对于未知领域的探索向来就是社会主义核心价值观中的一部分，作为新一代社会主义接班人，不把它们搞懂怎么支援祖国建设呢？

我们前边唠叨请求处理过程的时候提到过， `MySQL` 服务器上负责对表中数据的读取和写入工作的部分是 `存储引擎` ，而服务器又支持不同类型的存储引擎，比如 `InnoDB` 、 `MyISAM` 、 `Memory` 啥的，不同的存储引擎一般是由不同的人为实现不同的特性而开发的，真实数据在不同存储引擎中存放的格式一般是不同的，甚至有的存储引擎比如 `Memory` 都不用磁盘来存储数据，也就是说关闭服务器后表中的数据就消失了。由于 `InnoDB` 是 `MySQL` 默认的存储引擎，也是我们最常用到的存储引擎，我们也没有那么多时间去把各个存储引擎的内部实现都看一遍，所以本集要唠叨的是使用 `InnoDB` 作为存储引擎的数据存储结构，了解了一个存储引擎的数据存储结构之后，其他的存储引擎都是依葫芦画瓢，等我们用到了再说哈～

4.2 InnoDB页简介

InnoDB 是一个将表中的数据存储到磁盘上的存储引擎，所以即使关机后重启我们的数据还是存在的。而真正处理数据的过程是发生在内存中的，所以需要把磁盘中的数据加载到内存中，如果是处理写入或修改请求的话，还需要把内存中的内容刷新到磁盘上。而我们知道读写磁盘的速度非常慢，和内存读写差了几个数量级，所以当我们想从表中获取某些记录时， InnoDB 存储引擎需要一条一条的把记录从磁盘上读出来么？不，那样会慢死， InnoDB 采取的方式是：将数据划分为若干个页，以页作为磁盘和内存之间交互的基本单位，InnoDB中页的大小一般为 ***16*** KB。也就是在一般情况下，一次最少从磁盘中读取16KB的内容到内存中，一次最少把内存中的16KB内容刷新到磁盘中。

4.3 InnoDB行格式

我们平时是以记录为单位来向表中插入数据的，这些记录在磁盘上的存放方式也被称为 行格式 或者 记录格式 。设计 InnoDB 存储引擎的大叔们到现在为止设计了4种不同类型的 行格式 ，分别是 Compact 、 Redundant 、 Dynamic 和 Compressed 行格式，随着时间的推移，他们可能会设计出更多的行格式，但是不管怎么变，在原理上大体都是相同的。

4.3.1 指定行格式的语法

我们可以在创建或修改表的语句中指定 行格式 ：

```
CREATE TABLE 表名 (列的信息) ROW_FORMAT=行格式名称

ALTER TABLE 表名 ROW_FORMAT=行格式名称
```

比如我们在 xiaohaizi 数据库里创建一个演示用的表 record_format_demo ，可以这样指定它的 行格式 ：

```
mysql> USE xiaohaizi;
Database changed

mysql> CREATE TABLE record_format_demo (
    ->     c1 VARCHAR(10),
    ->     c2 VARCHAR(10) NOT NULL,
    ->     c3 CHAR(10),
    ->     c4 VARCHAR(10)
    -> ) CHARSET=ascii ROW_FORMAT=COMPACT;
Query OK, 0 rows affected (0.03 sec)
```

可以看到我们刚刚创建的这个表的 行格式 就是 Compact ，另外，我们还显式指定了这个表的字符集为 ascii ，因为 ascii 字符集只包括空格、标点符号、数字、大小写字母和一些不可见字符，所以我们的汉字是不能存到这个表里的。我们现在向这个表中插入两条记录：

```
mysql> INSERT INTO record_format_demo(c1, c2, c3, c4) VALUES('aaaa', 'bbb', 'cc', 'd'),
 ('eeee', 'fff', NULL, NULL);
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

现在表中的记录就是这个样子的：

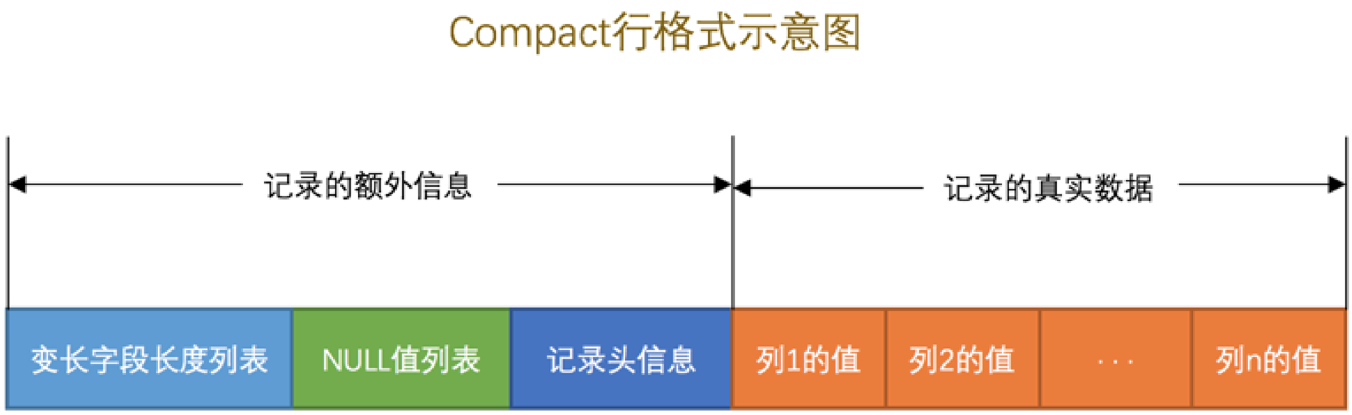
```
mysql> SELECT * FROM record_format_demo;
+------+-----+------+------+
| c1   | c2  | c3   | c4   |
+------+-----+------+------+
| aaaa | bbb | cc   | d    |
| eeee | fff | NULL | NULL |
+------+-----+------+------+
2 rows in set (0.00 sec)

mysql>
```

演示表的内容也填充好了，现在我们就来看看各个行格式下的存储方式到底有啥不同吧～

4.3.2 COMPACT行格式

废话不多说，直接看图：



大家从图中可以看出来，一条完整的记录其实可以被分为 记录的额外信息 和 记录的真实数据 两大部分，下边我们详细看一下这两部分的组成。

4.3.2.1 记录的额外信息

这部分信息是服务器为了描述这条记录而不得不额外添加的一些信息，这些额外信息分为3类，分别是 变长字段长度列表 、 NULL值列表 和 记录头信息 ，我们分别看一下。

变长字段长度列表

我们知道 MySQL 支持一些变长的数据类型，比如 VARCHAR(M) 、 VARBINARY(M) 、各种 TEXT 类型，各种 BLOB 类型，我们也可以把拥有这些数据类型的列称为 变长字段 ，变长字段中存储多少字节的数据是不固定的，所以我们在存储真实数据的时候需要顺便把这些数据占用的字节数也存起来，这样才不至于把 MySQL 服务器搞懵，所以这些变长字段占用的存储空间分为两部分：

1. 真正的数据内容
2. 占用的字节数

在 Compact 行格式中，把所有变长字段的真实数据占用的字节长度都存放在记录的开头部位，从而形成一个变长字段长度列表，各变长字段数据占用的字节数按照列的顺序逆序存放，我们再次强调一遍，是逆序存放！

我们拿 record_format_demo 表中的第一条记录来举个例子。因为 record_format_demo 表的 c1 、 c2 、 c4 列都是 VARCHAR(10) 类型的，也就是变长的数据类型，所以这三个列的值的长度都需要保存在记录开头处，因为 record_format_demo 表中的各个列都使用的是 ascii 字符集，所以每个字符只需要1个字节来进行编码，来看一下第一条记录各变长字段内容的长度：

列名	存储内容	内容长度（十进制表示）	内容长度（十六进制表示）
c1	'aaaa'	4	0x04
c2	'bbb'	3	0x03
c4	'd'	1	0x01

又因为这些长度值需要按照列的逆序存放，所以最后 变长字段长度列表 的字节串用十六进制表示的效果就是（各个字节之间实际上没有空格，用空格隔开只是方便理解）：

```
01 03 04
```

把这个字节串组成的 变长字段长度列表 填入上边的示意图中的效果就是：



由于第一行记录中 c1 、 c2 、 c4 列中的字符串都比较短，也就是说内容占用的字节数比较小，用1个字节就可以表示，但是如果变长列的内容占用的字节数比较多，可能就需要用2个字节来表示。具体用1个还是2个字节来表示真实数据占用的字节数， InnoDB 有它的一套规则，我们首先声明一下 W 、 M 和 L 的意思：

1. 假设某个字符集中表示一个字符最多需要使用的字节数为 W ，也就是使用 SHOW CHARSET 语句的结果中的 Maxlen 列，比方说 utf8 字符集中的 W 就是 3 ， gbk 字符集中的 W 就是 2 ， ascii 字符集中的 W 就是 1 。
2. 对于变长类型 VARCHAR(M) 来说，这种类型表示能存储最多 M 个字符（注意是字符不是字节），所以这个类型能表示的字符串最多占用的字节数就是 M×W 。
3. 假设它实际存储的字符串占用的字节数是 L 。

所以确定使用1个字节还是2个字节表示真正字符串占用的字节数的规则就是这样：

- 如果 M×W <= 255 ，那么使用1个字节来表示真正字符串占用的字节数。

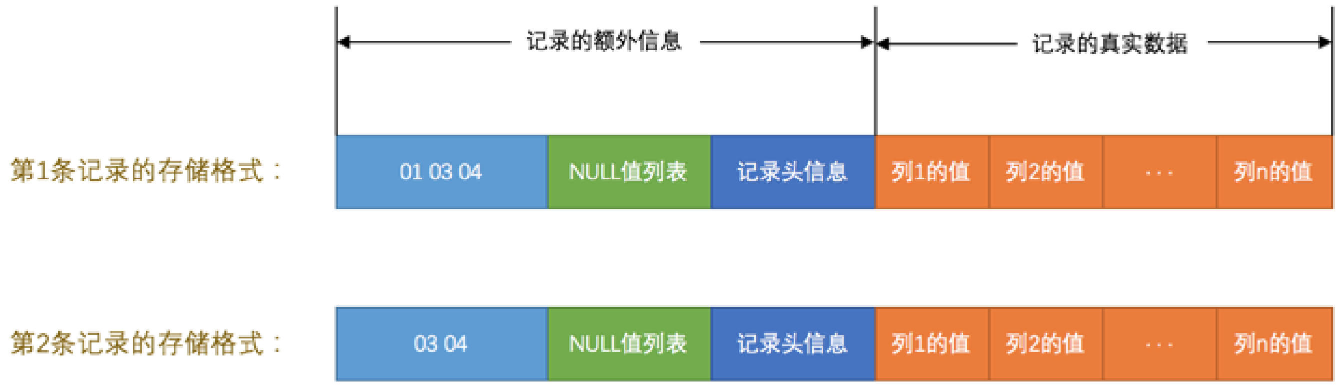
  > 也就是说InnoDB在读记录的变长字段长度列表时先查看表结构，如果某个变长字段允许存储的最大字节数不大于255时，可以认为只使用1个字节来表示真正字符串占用的字节数。

- 如果 M×W > 255 ，则分为两种情况：
  - 如果 L <= 127 ，则用1个字节来表示真正字符串占用的字节数。
  - 如果 L > 127 ，则用2个字节来表示真正字符串占用的字节数。

    > InnoDB在读记录的变长字段长度列表时先查看表结构，如果某个变长字段允许存储的最大字节数大于255时，该怎么区分它正在读的某个字节是一个单独的字段长度还是半个字段长度呢？设计InnoDB的大叔使用该字节的第一个二进制位作为标志位：如果该字节的第一个位为0，那该字节就是一个单独的字段长度（使用一个字节表示不大于127的二进制的第一个位都为0），如果该字节的第一个位为1，那该字节就是半个字段长度。
    >
    > 对于一些占用字节数非常多的字段，比方说某个字段长度大于了16KB，那么如果该记录在单个页面中无法存储时，InnoDB会把一部分数据存放到所谓的溢出页中（我们后边会唠叨），在变长字段长度列表处只存储留在本页面中的长度，所以使用两个字节也可以存放下来。

总结一下就是说：如果该可变字段允许存储的最大字节数（ M×W ）超过255字节并且真实存储的字节数（ L ）超过127字节，则使用2个字节，否则使用1个字节。

另外需要注意的一点是，变长字段长度列表中只存储值为 ***非NULL*** 的列内容占用的长度，值为 ***NULL*** 的列的长度是不储存的 。也就是说对于第二条记录来说，因为 c4 列的值为 NULL ，所以第二条记录的 变长字段长度列表 只需要存储 c1 和 c2 列的长度即可。其中 c1 列存储的值为 'eeee' ，占用的字节数为 4 ， c2 列存储的值为 'fff' ，占用的字节数为 3 。数字 4 可以用1个字节表示， 3 也可以用1个字节表示，所以整个 变长字段长度列表 共需2个字节。填充完 变长字段长度列表 的两条记录的对比图如下：



> 小贴士：
> 并不是所有记录都有这个 变长字段长度列表 部分，比方说表中所有的列都不是变长的数据类型的话，这一部分就不需要有。

NULL值列表

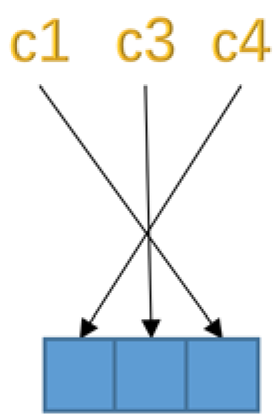
我们知道表中的某些列可能存储 NULL 值，如果把这些 NULL 值都放到 记录的真实数据 中存储会很占地方，所以 Compact 行格式把这些值为 NULL 的列统一管理起来，存储到 NULL 值列表中，它的处理过程是这样的：

1. 首先统计表中允许存储 NULL 的列有哪些。

   我们前边说过，主键列、被 NOT NULL 修饰的列都是不可以存储 NULL 值的，所以在统计的时候不会把这些列算进去。比方说表 record_format_demo 的3个列 c1 、 c3 、 c4 都是允许存储 NULL 值的，而 c2 列是被 NOT NULL 修饰，不允许存储 NULL 值。
2. 如果表中没有允许存储 ***NULL*** 的列，则 *NULL值列表* 也不存在了，否则将每个允许存储 NULL 的列对应一个二进制位，二进制位按照列的顺序逆序排列，二进制位表示的意义如下：

   - 二进制位的值为 1 时，代表该列的值为 NULL 。
   - 二进制位的值为 0 时，代表该列的值不为 NULL 。

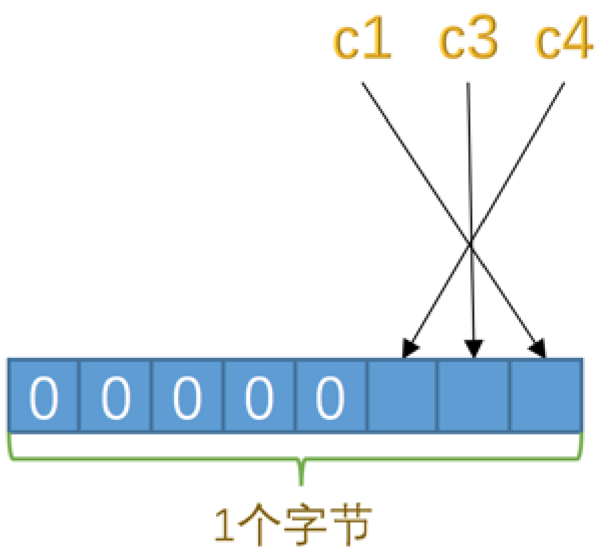
   因为表 record_format_demo 有3个值允许为 NULL 的列，所以这3个列和二进制位的对应关系就是这样：



再一次强调，二进制位按照列的顺序逆序排列，所以第一个列 c1 和最后一个二进制位对应。

3. MySQL 规定 NULL值列表 必须用整数个字节的位表示，如果使用的二进制位个数不是整数个字节，则在字节的高位补 0 。

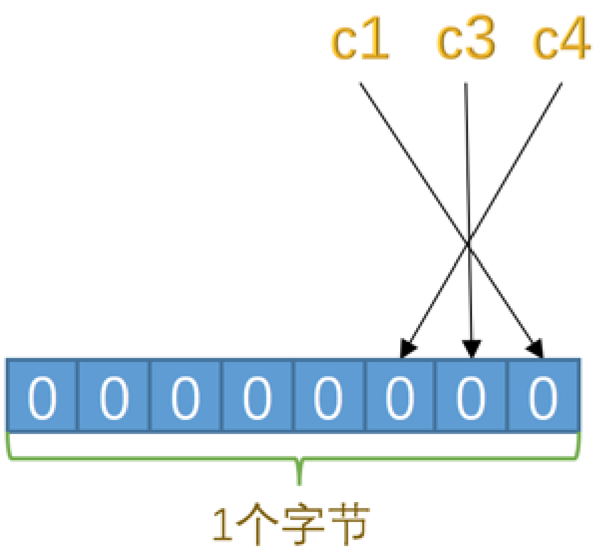
表 record_format_demo 只有3个值允许为 NULL 的列，对应3个二进制位，不足一个字节，所以在字节的高位补 0 ，效果就是这样：



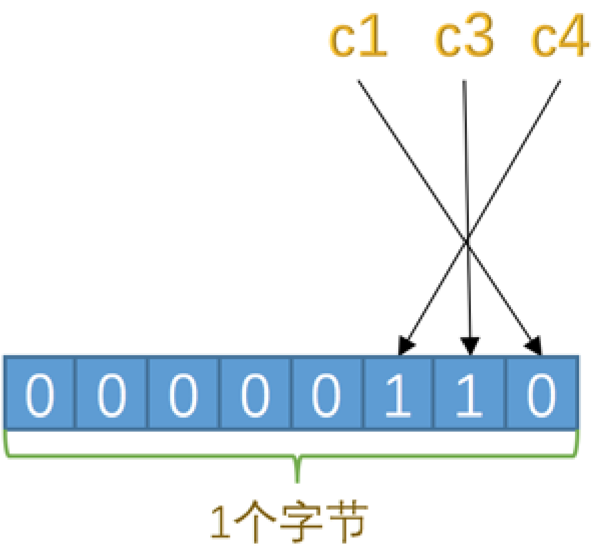
以此类推，如果一个表中有9个允许为 NULL ，那这个记录的 NULL 值列表部分就需要2个字节来表示了。

知道了规则之后，我们再返回头看表 record_format_demo 中的两条记录中的 NULL值列表 应该怎么储存。因为只有 c1 、 c3 、 c4 这3个列允许存储 NULL 值，所以所有记录的 NULL值列表 只需要一个字节。

- 对于第一条记录来说， c1 、 c3 、 c4 这3个列的值都不为 NULL ，所以它们对应的二进制位都是 0 ，画个图就是这样：

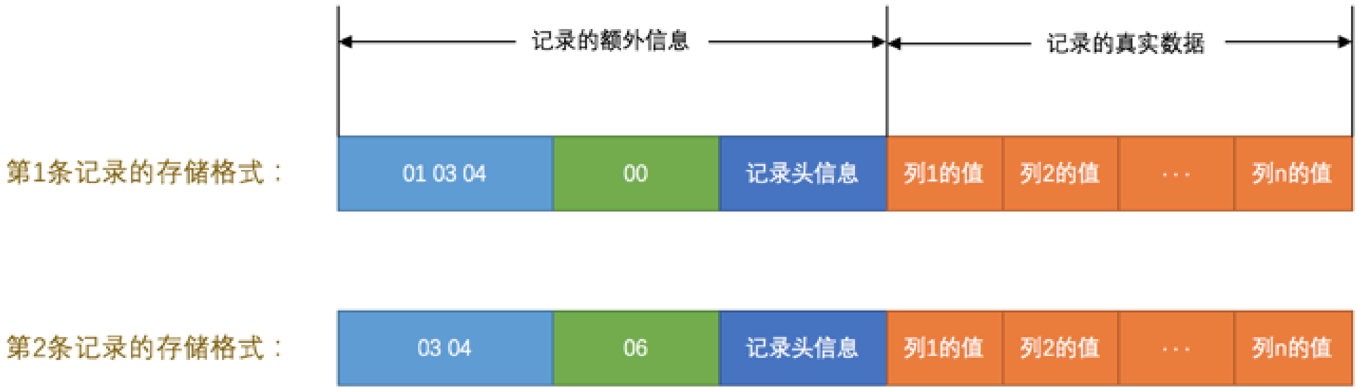


  所以第一条记录的 NULL值列表 用十六进制表示就是： 0x00 。
- 对于第二条记录来说， c1 、 c3 、 c4 这3个列中 c3 和 c4 的值都为 NULL ，所以这3个列对应的二进制位的情况就是：



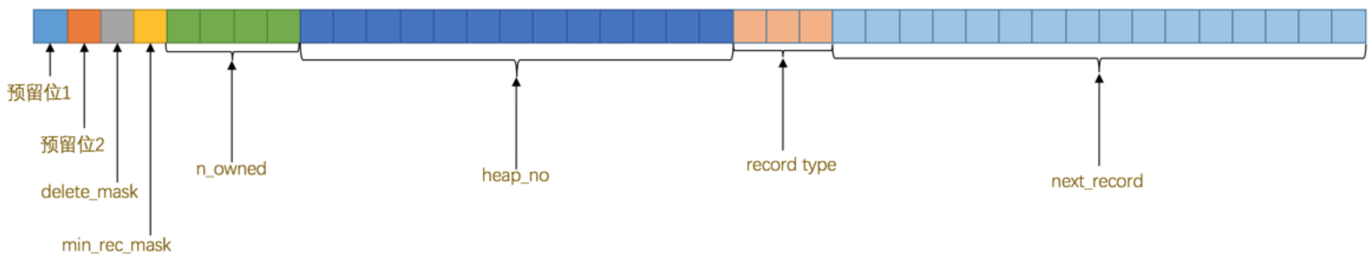
所以第二条记录的 NULL值列表 用十六进制表示就是： 0x06 。

所以这两条记录在填充了 NULL值列表 后的示意图就是这样：



记录头信息

除了 变长字段长度列表 、 NULL值列表 之外，还有一个用于描述记录的 记录头信息 ， 它是由固定的 5 个字节组成。 5 个字节也就是 40 个二进制位，不同的位代表不同的意思，如图：



这些二进制位代表的详细信息如下表：

名称	大小（单位：bit）	描述
预留位1	1	没有使用
预留位2	1	没有使用

名称	大小（单位：bit）	描述
delete_mask	1	标记该记录是否被删除
min_rec_mask	1	B+树的每层非叶子节点中的最小记录都会添加该标记
n_owned	4	表示当前记录拥有的记录数
heap_no	13	表示当前记录在记录堆的位置信息
record_type	3	表示当前记录的类型， 0 表示普通记录， 1 表示B+树非叶子节点记录， 2 表示最小记录， 3 表示最大记录
next_record	16	表示下一条记录的相对位置

大家不要被这么多的属性和陌生的概念给吓着，我这里只是为了内容的完整性把这些位代表的意思都写了出来，现在没必要把它们的意思都记住，记住也没啥用，现在只需要看一遍混个脸熟，等之后用到这些属性的时候我们再回过头来看。

因为我们并不清楚这些属性详细的用法，所以这里就不分析各个属性值是怎么产生的了，之后我们遇到会详细看的。所以我们现在直接看一下 record_format_demo 中的两条记录的 头信息 分别是什么：

> 小贴士：
> 再一次强调，大家如果看不懂记录头信息里各个位代表的概念千万别纠结，我们后边会说的～

4.3.2.2 记录的真实数据

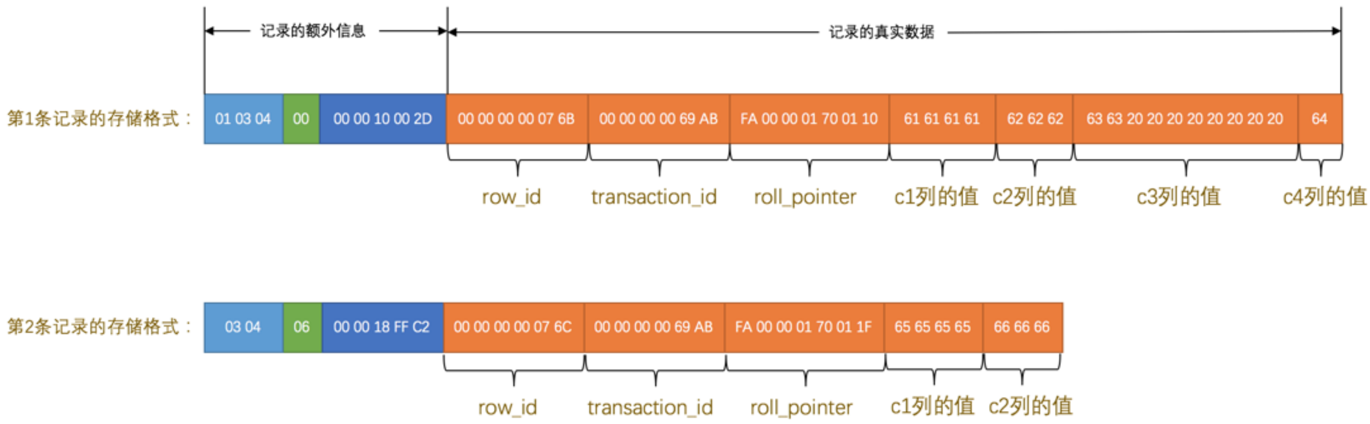
对于 record_format_demo 表来说， 记录的真实数据 除了 c1 、 c2 、 c3 、 c4 这几个我们自己定义的列的数据以外， MySQL 会为每个记录默认的添加一些列（也称为 隐藏列 ），具体的列如下：

列名	是否必须	占用空间	描述
row_id	否	6 字节	行ID，唯一标识一条记录
transaction_id	是	6 字节	事务ID
roll_pointer	是	7 字节	回滚指针

> 小贴士：
> 实际上这几个列的真正名称其实是：DB_ROW_ID、DB_TRX_ID、DB_ROLL_PTR，我们为了美观才写成了row_id、transaction_id和roll_pointer。

这里需要提一下 InnoDB 表对主键的生成策略：优先使用用户自定义主键作为主键，如果用户没有定义主键，则选取一个 Unique 键作为主键，如果表中连 Unique 键都没有定义的话，则 InnoDB 会为表默认添加一个名为 row_id 的隐藏列作为主键。所以我们从上表中可以看出：InnoDB存储引擎会为每条记录都添加 ***transaction_id*** 和 ***roll_pointer*** 这两个列，但是 ***row_id*** 是可选的（在没有自定义主键以及Unique键的情况下才会添加该列）。这些隐藏列的值不用我们操心， InnoDB 存储引擎会自己帮我们生成的。

因为表 record_format_demo 并没有定义主键，所以 MySQL 服务器会为每条记录增加上述的3个列。现在看一下加上 记录的真实数据 的两个记录长什么样吧：

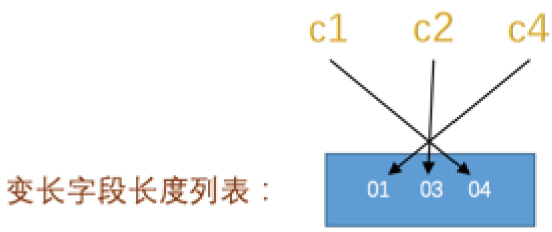


看这个图的时候我们需要注意几点：

1. 表 record_format_demo 使用的是 ascii 字符集，所以 0x61616161 就表示字符串 'aaaa'， 0x626262 就表示字符串 'bbb'，以此类推。
2. 注意第1条记录中 c3 列的值，它是 CHAR(10) 类型的，它实际存储的字符串是： 'cc'，而 ascii 字符集中的字节表示是 '0x6363'，虽然表示这个字符串只占用了2个字节，但整个 c3 列仍然占用了10个字节的空间，除真实数据以外的8个字节的统统都用空格字符填充，空格字符在 ascii 字符集的表示就是 0x20 。
3. 注意第2条记录中 c3 和 c4 列的值都为 NULL ，它们被存储在了前边的 NULL值列表 处，在记录的真实数据处就不再冗余存储，从而节省存储空间。

4.3.2.3 CHAR(M)列的存储格式

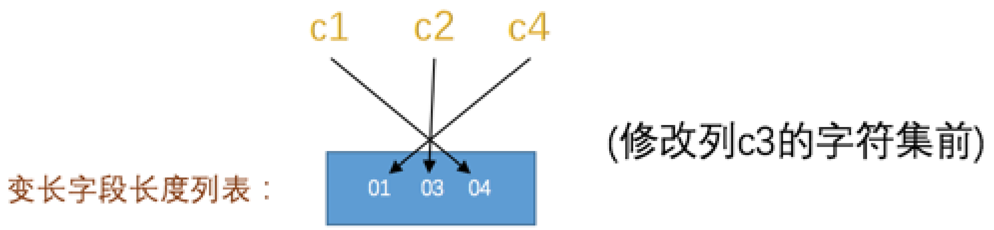
record_format_demo 表的 c1 、 c2 、 c4 列的类型是 VARCHAR(10) ，而 c3 列的类型是 CHAR(10) ，我们说在 Compact 行格式下只会把变长类型的列的长度逆序存到 变长字段长度列表 中，就像这样：

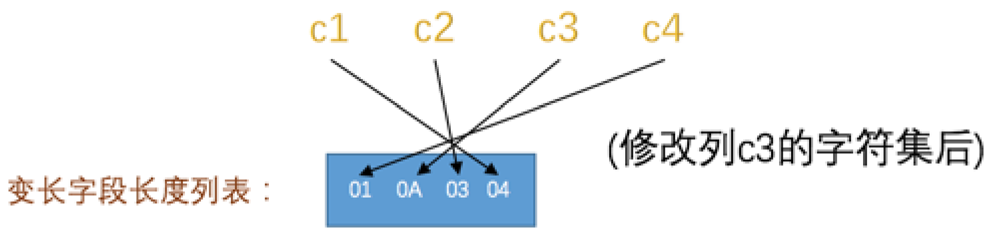


但是这只是因为我们的 record_format_demo 表采用的是 ascii 字符集，这个字符集是一个定长字符集，也就是说表示一个字符采用固定的一个字节，如果采用变长的字符集（也就是表示一个字符需要的字节数不确定，比如 gbk 表示一个字符要1~~2个字节、 utf8 表示一个字符要1~~3个字节等）的话， c3 列的长度也会被存储到 变长字段长度列表 中，比如我们修改一下 record_format_demo 表的字符集：

```
mysql> ALTER TABLE record_format_demo MODIFY COLUMN c3 CHAR(10) CHARACTER SET utf8;
Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

修改该列字符集后记录的 变长字段长度列表 也发生了变化，如图：





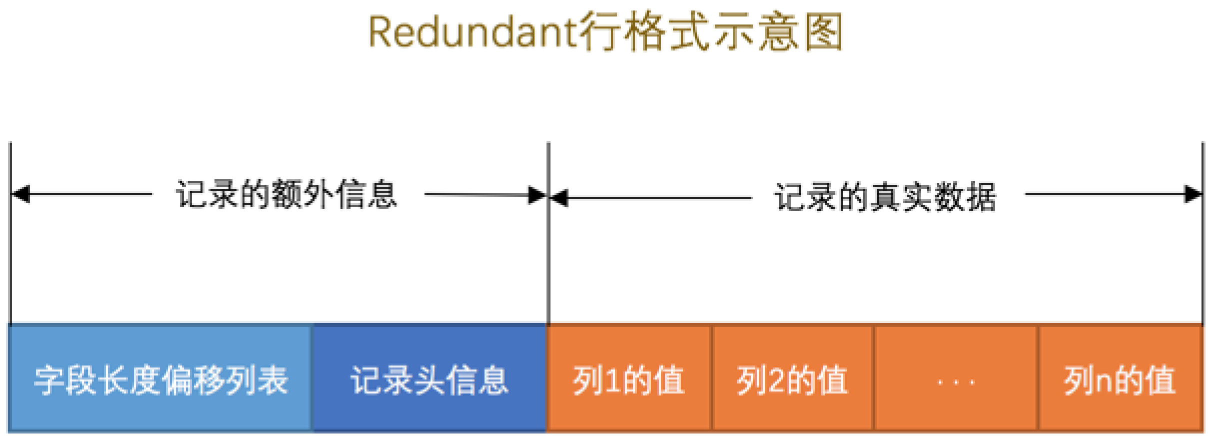
这就意味着：对于 ***CHAR(M)*** 类型的列来说，当列采用的是定长字符集时，该列占用的字节数不会被加到变长字段长度列表，而如果采用变长字符集时，该列占用的字节数也会被加到变长字段长度列表。

另外有一点还需要注意，变长字符集的 CHAR(M) 类型的列要求至少占用 M 个字节，而 VARCHAR(M) 却没有这个要求。比方说对于使用 utf8 字符集的 CHAR(10) 的列来说，该列存储的数据字节长度的范围是10～30个字节。即使我们向该列中存储一个空字符串也会占用 10 个字节，这是怕将来更新该列的值的字节长度大于原有值的字节长度而小于10个字节时，可以在该记录处直接更新，而不是在存储空间中重新分配一个新的记录空间，导致原有的记录空间成为所谓的碎片。（这里你感受到设计 Compact 行格式的大叔既想节省存储空间，又不想更新 CHAR(M) 类型的列产生碎片时的纠结心情了吧。）

4.3.3 Redundant行格式

其实知道了 Compact 行格式之后，其他的行格式就是依葫芦画瓢了。我们现在要介绍的 Redundant 行格式是 MySQL5.0 之前用的一种行格式，也就是说它已经非常老了，但是本着知识完整性的角度还是要提一下，大家乐呵乐呵的看就好。

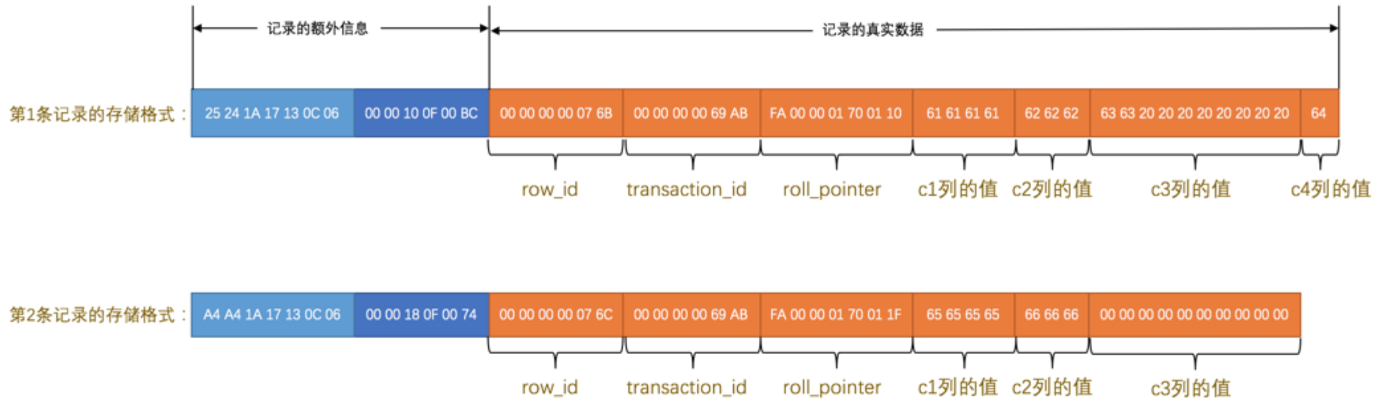
画个图展示一下 Redundant 行格式的全貌：



现在我们把表 record_format_demo 的行格式修改为 Redundant ：

```
mysql> ALTER TABLE record_format_demo ROW_FORMAT=Redundant;
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

为了方便大家理解和节省篇幅，我们直接把表 record_format_demo 在 Redundant 行格式下的两条记录的真实存储数据提供出来，之后我们着重分析两种行格式的不同即可。



下边我们从各个方面看一下 Redundant 行格式有什么不同的地方：

- 字段长度偏移列表

  注意 Compact 行格式的开头是 变长字段长度列表 ，而 Redundant 行格式的开头是 字段长度偏移列表 ，与 变长字段长度列表 有两处不同：

  - 没有了变长两个字，意味着 Redundant 行格式会把该条记录中所有列 （包括 隐藏列 ）的长度信息都按照逆序存储到 字段长度偏移列表 。
  - 多了个偏移两个字，这意味着计算列值长度的方式不像 Compact 行格式那么直观，它是采用两个相邻数值的差值来计算各个列值的长度。

  比如第一条记录的 字段长度偏移列表 就是：

  ```
  25 24 1A 17 13 0C 06
  ```

  因为它是逆序排放的，所以按照列的顺序排列就是：

  ```
  06 0C 13 17 1A 24 25
  ```

  按照两个相邻数值的差值来计算各个列值的长度的意思就是：

第一列(`row_id`)的长度就是 0x06个字节，也就是6个字节。

第二列(`transaction_id`)的长度就是 (0x0C - 0x06)个字节，也就是6个字节。

第三列(`roll_pointer`)的长度就是 (0x13 - 0x0C)个字节，也就是7个字节。

第四列(`c1`)的长度就是 (0x17 - 0x13)个字节，也就是4个字节。

第五列(`c2`)的长度就是 (0x1A - 0x17)个字节，也就是3个字节。

第六列(`c3`)的长度就是 (0x24 - 0x1A)个字节，也就是10个字节。

第七列(`c4`)的长度就是 (0x25 - 0x24)个字节，也就是1个字节。

- 记录头信息

  Redundant 行格式的记录头信息占用 6 字节， 48 个二进制位，这些二进制位代表的意思如下：

  |名称|大小（单位：bit）|描述| |:--:|:--:|:--:| | 预留位1 | 1 |没有使用| | 预留位2 | 1 |没有使用| | delete_mask | 1 |标记该记录是否被删除| | min_rec_mask | 1 |B+树的每层非叶子节点中的最小记录都会添加该标记| | n_owned | 4 |表示当前记录拥有的记录数| | heap_no | 13 |表示当前记录在页面堆的位置信息| | n_field | 10 |表示记录中列的数量| | 1byte_offs_flag | 1 |标记字段长度偏移列表中每个列对应的偏移量是使用1字节还是2字节表示的| | next_record | 16 |表示下一条记录的相对位置|

  第一条记录中的头信息是：

  ```
  00 00 10 0F 00 BC
  ```

  根据这六个字节可以计算出各个属性的值，如下：

  ```
  预留位1：0x00
  预留位2：0x00
  delete_mask: 0x00
  min_rec_mask: 0x00
  n_owned: 0x00
  heap_no: 0x02
  n_field: 0x07
  1byte_offs_flag: 0x01
  next_record:0xBC
  ```

  与 Compact 行格式的记录头信息对比来看，有两处不同：
  - Redundant 行格式多了 n_field 和 1byte_offs_flag 这两个属性。
  - Redundant 行格式没有 record_type 这个属性。
- 1byte_offs_flag 的值是怎么选择的

  字段长度偏移列表 实质上是存储每个列中的值占用的空间在 记录的真实数据 处结束的位置，还是拿 record_format_demo 第一条记录为例， 0x06 代表第一个列在 记录的真实数据 第6个字节处结束， 0x0C 代表第二个列在 记录的真实数据 第12个字节处结束， 0x13 代表第三个列在 记录的真实数据 第19个字节处结束，等等等等，最后一个列对应的偏移量值为 0x25 ，也就意味着最后一个列在 记录的真实数据 第37个字节处结束，也就意味着整条记录的 真实数据 实际上占用 37 个字节。

  我们前边说过每个列对应的偏移量可以占用1个字节或者2个字节来存储，那到底什么时候用1个字节，什么时候用2个字节呢？其实是根据该条 Redundant 行格式 记录的真实数据 占用的总大小来判断的：
  - 当记录的真实数据占用的字节数不大于127（十六进制 0x7F ，二进制 01111111 ）时，每个列对应的偏移量占用1个字节。

> 小贴士：
> 如果整个记录的真实数据占用的存储空间都不大于127个字节，那么每个列对应的偏移量值肯定也就不大于127，也就可以使用1个字节来表示喽。

- 当记录的真实数据占用的字节数大于127，但不大于32767（十六进制 `0x7FFF` ，二进制 `0111111111111111` ）时，每个列对应的偏移量占用2个字节。
- 有没有记录的真实数据大于32767的情况呢？有，不过此时的记录已经存放到了溢出页中，在本页中只保留前 `768` 个字节和20个字节的溢出页面地址（当然这20个字节中还记录了一些别的信息）。因为 `字段长度偏移列表` 处只需要记录每个列在本页面中的偏移就好了，所以每个列使用2个字节来存储偏移量就够了。

  大家可以看出来，设计 `Redundant` 行格式的大叔还是比较简单粗暴的，直接使用整个 `记录的真实数据` 长度来决定使用1个字节还是2个字节存储列对应的偏移量。只要整条记录的真实数据占用的存储空间大小大于127，即使第一个列的值占用存储空间小于127，那对不起，也需要使用2个字节来表示该列对应的偏移量。简单粗暴，就是这么简单粗暴（所以这种行格式有些过时了~）。

  > 小贴士：
  > 大家有没有疑惑，一个字节能表示的范围是0～255，为啥在记录的真实数据占用的存储空间大于127时就采用2个字节表示各个列的偏移量呢？稍安勿躁，后边马上揭晓。

  为了在解析记录时知道每个列的偏移量是使用1个字节还是2个字节表示的，设计 `Redundant` 行格式的大叔特意在 `记录头信息` 里放置了一个称之为 `1byte_offs_flag` 的属性：
- 当它的值为1时，表明使用1个字节存储。
- 当它的值为0时，表明使用2个字节存储。

* `Redundant` 行格式中 `NULL` 值的处理

  因为 `Redundant` 行格式并没有 `NULL值列表` ，所以设计 `Redundant` 行格式的大叔在 `字段长度偏移列表` 中的各个列对应的偏移量处做了一些特殊处理 —— 将列对应的偏移量值的第一个比特位作为是否为 `NULL` 的依据，该比特位也可以被称之为 `NULL比特位` 。也就是说在解析一条记录的某个列时，首先看一下该列对应的偏移量的 `NULL比特位` 是不是为 `1` ，如果为 `1` ，那么该列的值就是 `NULL` ，否则不是 `NULL` 。

  这也就解释了上边介绍为什么只要记录的真实数据大于127（十六进制 `0x7F` ，二进制 `01111111` ）时，就采用2个字节来表示一个列对应的偏移量，主要是第一个比特位是所谓的 `NULL比特位` ，用来标记该列的值是否为 `NULL` 。

  但是还有一点要注意，对于值为 `NULL` 的列来说，该列的类型是否为定长类型决定了 `NULL` 值的实际存储方式，我们接下来分析一下 `record_format_demo` 表的第二条记录，它对应的 `字段长度偏移列表` 如下：

  ```
  A4 A4 1A 17 13 0C 06
  ```

  按照列的顺序排放就是：

  ```
  06 0C 13 17 1A A4 A4
  ```

  我们分情况看一下：
  - 如果存储 `NULL` 值的字段是定长类型的，比方说 `CHAR(M)` 数据类型的，则 `NULL` 值也将占用记录的真实数据部分，并把该字段对应的数据使用 `0x00` 字节填充。

    如图第二条记录的 `c3` 列的值是 `NULL` ，而 `c3` 列的类型是 `CHAR(10)` ，占用记录的真实数据部分10字节，所以我们看到在 `Redundant` 行格式中使用 `0x00000000000000000000` 来表示 `NULL` 值。

    另外， `c3` 列对应的偏移量为 `0xA4` ，它对应的二进制实际是： `10100100` ，可以看到最高位为 `1` ，意味着该列的值是 `NULL` 。将最高位去掉后的值变成了 `0100100` ，对应的十进制值为 `36` ，而 `c2` 列对应的偏移量为 `0x1A` ，也就是十进制的 `26` 。 `36 - 26 = 10` ，也就是说最终 `c3` 列占用的存储空间为10个字节。
  - 如果该存储 `NULL` 值的字段是变长数据类型的，则不在 `记录的真实数据` 处占用任何存储空间。

比如 record_format_demo 表的 c4 列是 VARCHAR(10) 类型的， VARCHAR(10) 是一个变长数据类型， c4 列对应的偏移量为 0xA4 ，与 c3 列对应的偏移量相同，这也就意味着它的值也为 NULL ，将 0xA4 的最高位去掉后对应的十进制值也是 36 ， 36 - 36 = 0 ，也就意味着 c4 列本身不占用任何 记录的实际数据 处的空间。

除了以上的几点之外， Redundant 行格式和 Compact 行格式还是大致相同的。

4.3.3.1 CHAR(M)列的存储格式

我们知道 Compact 行格式在 CHAR(M) 类型的列中存储数据的时候还挺麻烦，分变长字符集和定长字符集的情况，而在 Redundant 行格式中十分干脆，不管该列使用的字符集是啥，只要是使用 CHAR(M) 类型，占用的真实数据空间就是该字符集表示一个字符最多需要的字节数和 M 的乘积。比方说使用 utf8 字符集的 CHAR(10) 类型的列占用的真实数据空间始终为 30 个字节，使用 gbk 字符集的 CHAR(10) 类型的列占用的真实数据空间始终为 20 个字节。由此可以看出来，使用 Redundant 行格式的 CHAR(M) 类型的列是不会产生碎片的。

4.3.4 行溢出数据

4.3.4.1 VARCHAR(M)最多能存储的数据

我们知道对于 VARCHAR(M) 类型的列最多可以占用 65535 个字节。其中的 M 代表该类型最多存储的字符数量，如果我们使用 ascii 字符集的话，一个字符就代表一个字节，我们看看 VARCHAR(65535) 是否可用：

```
mysql> CREATE TABLE varchar_size_demo(
    ->     c VARCHAR(65535)
    -> ) CHARSET=ascii ROW_FORMAT=Compact;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not
 counting BLOBs, is 65535. This includes storage overhead, check the manual. You have to c
hange some columns to TEXT or BLOBs
mysql>
```

从报错信息里可以看出， MySQL 对一条记录占用的最大存储空间是有限制的，除了 BLOB 或者 TEXT 类型的列之外，其他所有的列（不包括隐藏列和记录头信息）占用的字节长度加起来不能超过 65535 个字节。所以 MySQL 服务器建议我们把存储类型改为 TEXT 或者 BLOB 的类型。这个 65535 个字节除了列本身的数据之外，还包括一些其他的数据（ storage overhead ），比如说我们为了存储一个 VARCHAR(M) 类型的列，其实需要占用3部分存储空间：

- 真实数据
- 真实数据占用字节的长度
- NULL 值标识，如果该列有 NOT NULL 属性则可以没有这部分存储空间

如果该 VARCHAR 类型的列没有 NOT NULL 属性，那最多只能存储 65532 个字节的数据，因为真实数据的长度可能占用2个字节， NULL 值标识需要占用1个字节：

```
mysql> CREATE TABLE varchar_size_demo(
    ->      c VARCHAR(65532)
    -> ) CHARSET=ascii ROW_FORMAT=Compact;
Query OK, 0 rows affected (0.02 sec)
```

如果 VARCHAR 类型的列有 NOT NULL 属性，那最多只能存储 65533 个字节的数据，因为真实数据的长度可能占用2个字节，不需要 NULL 值标识：

```
mysql> DROP TABLE varchar_size_demo;
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE varchar_size_demo(
    ->      c VARCHAR(65533) NOT NULL
    -> ) CHARSET=ascii ROW_FORMAT=Compact;
Query OK, 0 rows affected (0.02 sec)
```

如果 VARCHAR(M) 类型的列使用的不是 ascii 字符集，那会怎么样呢？来看一下：

```
mysql> DROP TABLE varchar_size_demo;
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE varchar_size_demo(
    ->       c VARCHAR(65532)
    -> ) CHARSET=gbk ROW_FORMAT=Compact;
ERROR 1074 (42000): Column length too big for column 'c' (max = 32767); use BLOB or TEXT i
nstead

mysql> CREATE TABLE varchar_size_demo(
    ->       c VARCHAR(65532)
    -> ) CHARSET=utf8 ROW_FORMAT=Compact;
ERROR 1074 (42000): Column length too big for column 'c' (max = 21845); use BLOB or TEXT i
nstead
```

从执行结果中可以看出，如果 VARCHAR(M) 类型的列使用的不是 ascii 字符集，那 M 的最大取值取决于该字符集表示一个字符最多需要的字节数。在列的值允许为 NULL 的情况下， gbk 字符集表示一个字符最多需要 2 个字节，那在该字符集下， M 的最大取值就是 32766 （也就是：65532/2），也就是说最多能存储 32766 个字符； utf8 字符集表示一个字符最多需要 3 个字节，那在该字符集下， M 的最大取值就是 21844 ，就是说最多能存储 21844 （也就是：65532/3）个字符。

> 小贴士：
> 上述所言在列的值允许为NULL的情况下，gbk字符集下M的最大取值就是32766，utf8字符集下M的最大取值就是21844，这都是在表中只有一个字段的情况下说的，一定要记住一个行中的所有列（不包括隐藏列和记录头信息）占用的字节长度加起来不能超过65535个字节！

4.3.4.2 记录中的数据太多产生的溢出

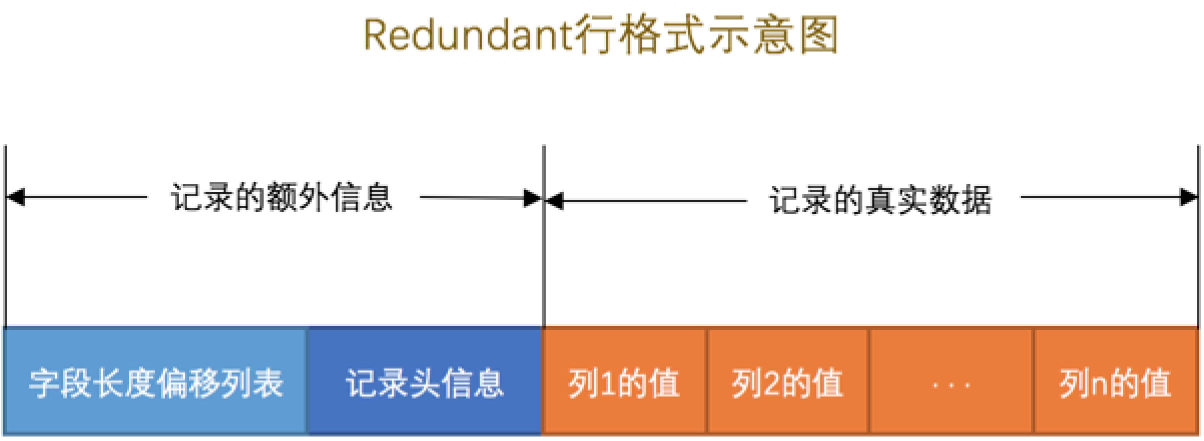
我们以 ascii 字符集下的 varchar_size_demo 表为例，插入一条记录：

```
mysql> CREATE TABLE varchar_size_demo(
    ->       c VARCHAR(65532)
    -> ) CHARSET=ascii ROW_FORMAT=Compact;
Query OK, 0 rows affected (0.01 sec)

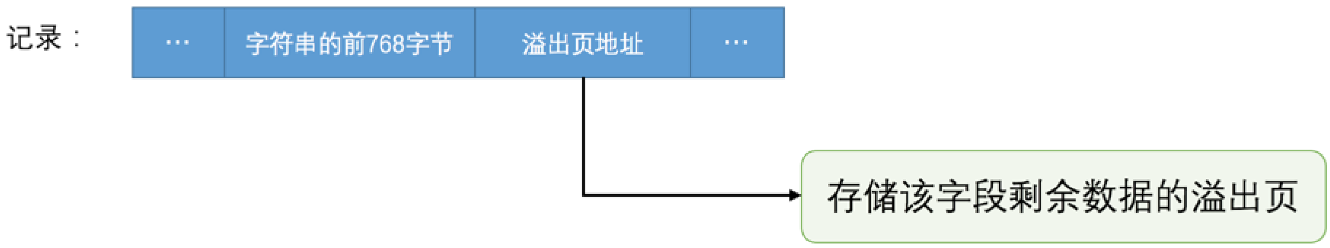
mysql> INSERT INTO varchar_size_demo(c) VALUES(REPEAT('a', 65532));
Query OK, 1 row affected (0.00 sec)
```

其中的 REPEAT('a', 65532) 是一个函数调用，它表示生成一个把字符 'a' 重复 65532 次的字符串。前边说过， MySQL 中磁盘和内存交互的基本单位是 页 ，也就是说 MySQL 是以 页 为基本单位来管理存储空间的，我们的记录都会被分配到某个 页 中存储。而一个页的大小一般是 16KB ，也就是 16384 字节，而一个 VARCHAR(M) 类型的列就最多可以存储 65532 个字节，这样就可能造成一个页存放不了一条记录的尴尬情况。

在 Compact 和 Reduntant 行格式中，对于占用存储空间非常大的列，在 记录的真实数据 处只会存储该列的一部分数据，把剩余的数据分散存储在几个其他的页中，然后 记录的真实数据 处用20个字节存储指向这些页的地址（当然这20个字节中还包括这些分散在其他页面中的数据的占用的字节数），从而可以找到剩余数据所在的页，如图所示：



从图中可以看出来，对于 Compact 和 Reduntant 行格式来说，如果某一列中的数据非常多的话，在本记录的真实数据处只会存储该列的前 768 个字节的数据和一个指向其他页的地址，然后把剩下的数据存放到其他页中，这个过程也叫做 行溢出 ，存储超出 768 字节的那些页面也被称为 溢出页 。画一个简图就是这样：



最后需要注意的是，不只是 ***VARCHAR(M)*** 类型的列，其他的 ***TEXT***、***BLOB*** 类型的列在存储数据非常多的时候也会发生 行溢出 。

4.3.4.3 行溢出的临界点

那发生 行溢出 的临界点是什么呢？也就是说在列存储多少字节的数据时就会发生 行溢出 ？

MySQL 中规定一个页中至少存放两行记录，至于为什么这么规定我们之后再说，现在看一下这个规定造成的影响。以上边的 varchar_size_demo 表为例，它只有一个列 c ，我们往这个表中插入两条记录，每条记录最少插入多少字节的数据才会 行溢出 的现象呢？这得分析一下页中的空间都是如何利用的。

- 每个页除了存放我们的记录以外，也需要存储一些额外的信息，乱七八糟的额外信息加起来需要 136 个字节的空间（现在只要知道这个数字就好了），其他的空间都可以被用来存储记录。
- 每个记录需要的额外信息是 27 字节。

  这27个字节包括下边这些部分：
  - 2个字节用于存储真实数据的长度
  - 1个字节用于存储列是否是NULL值
  - 5个字节大小的头信息
  - 6个字节的 row_id 列
  - 6个字节的 transaction_id 列
  - 7个字节的 roll_pointer 列

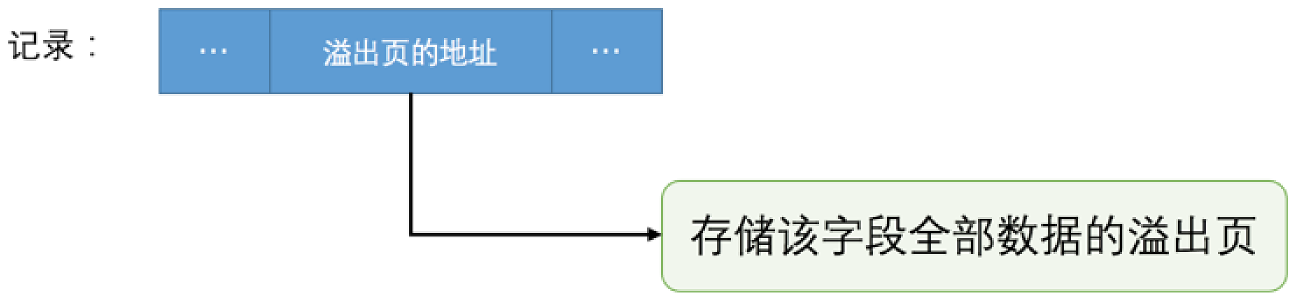
假设一个列中存储的数据字节数为n，那么发生 行溢出 现象时需要满足这个式子：

```
136 + 2×(27 + n) > 16384
```

求解这个式子得出的解是： n > 8098 。也就是说如果一个列中存储的数据不大于 8098 个字节，那就不会发生 行溢出 ，否则就会发生 行溢出 。不过这个 8098 个字节的结论只是针对只有一个列的 varchar_size_demo 表来说的，如果表中有多个列，那上边的式子和结论都需要改一改了，所以重点就是：你不用关注这个临界点是什么，只要知道如果我们想一个行中存储了很大的数据时，可能发生 行溢出 的现象。

4.3.5 Dynamic和Compressed行格式

下边要介绍另外两个行格式， Dynamic 和 Compressed 行格式，我现在使用的 MySQL 版本是 5.7 ，它的默认行格式就是 Dynamic ，这俩行格式和 Compact 行格式挺像，只不过在处理 行溢出 数据时有点儿分歧，它们不会在记录的真实数据处存储字段真实数据的前 768 个字节，而是把所有的字节都存储到其他页面中，只在记录的真实数据处存储其他页面的地址，就像这样：



Compressed 行格式和 Dynamic 不同的一点是， Compressed 行格式会采用压缩算法对页面进行压缩，以节省空间。

4.4 总结

1. 页是 MySQL 中磁盘和内存交互的基本单位，也是 MySQL 是管理存储空间的基本单位。
2. 指定和修改行格式的语法如下：

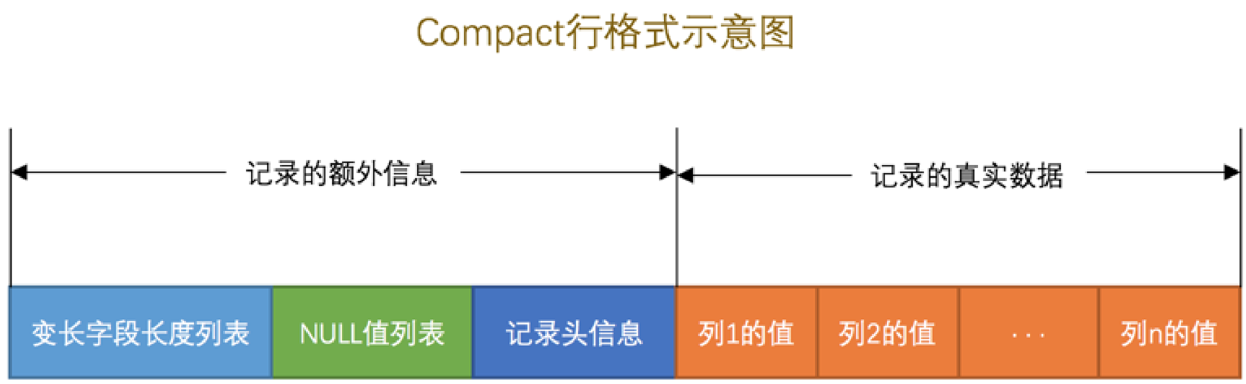
```
CREATE TABLE 表名 (列的信息) ROW_FORMAT=行格式名称

ALTER TABLE 表名 ROW_FORMAT=行格式名称
```

3. InnoDB 目前定义了4种行格式

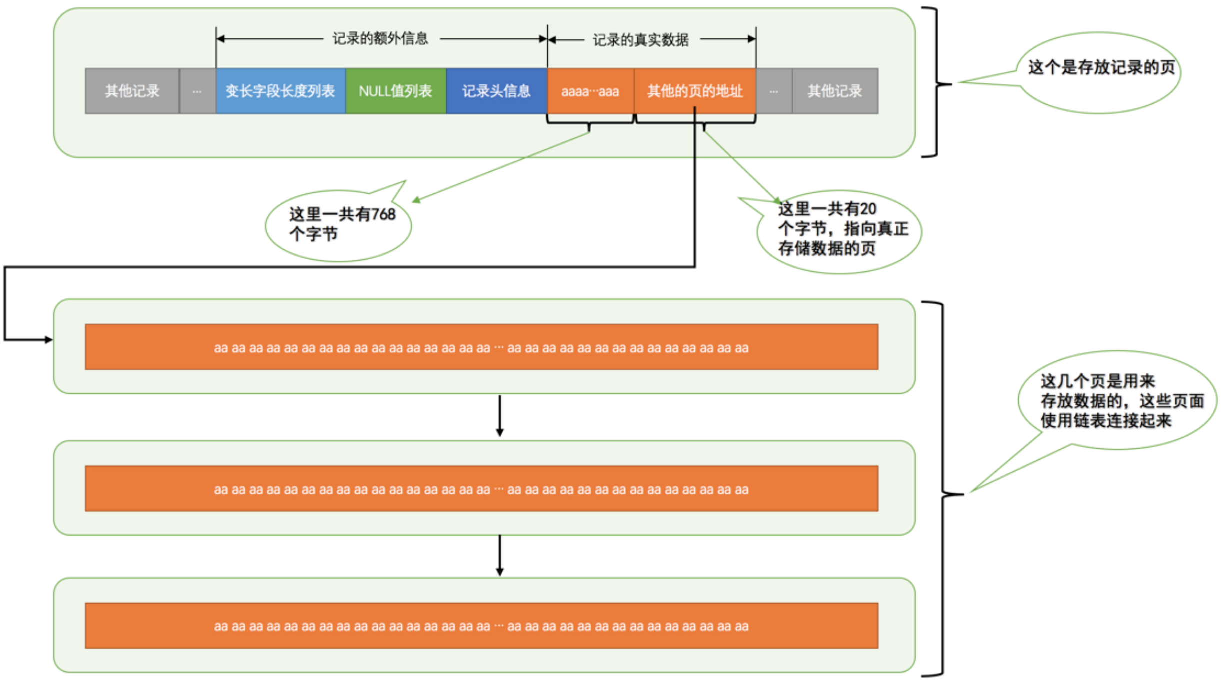
- COMPACT行格式

  具体组成如图：



- Redundant行格式

  具体组成如图：



- Dynamic和Compressed行格式

  这两种行格式类似于 COMPACT行格式 ，只不过在处理行溢出数据时有点儿分歧，它们不会在记录的真实数据处存储字符串的前768个字节，而是把所有的字节都存储到其他页面中，只在记录的真实数据处存储其他页面的地址。

  另外， Compressed 行格式会采用压缩算法对页面进行压缩。

4. 一个页一般是 16KB ，当记录中的数据太多，当前页放不下的时候，会把多余的数据存储到其他页中，这种现象称为 行溢出 。

5 第5章 盛放记录的大盒子-InnoDB数据页结构

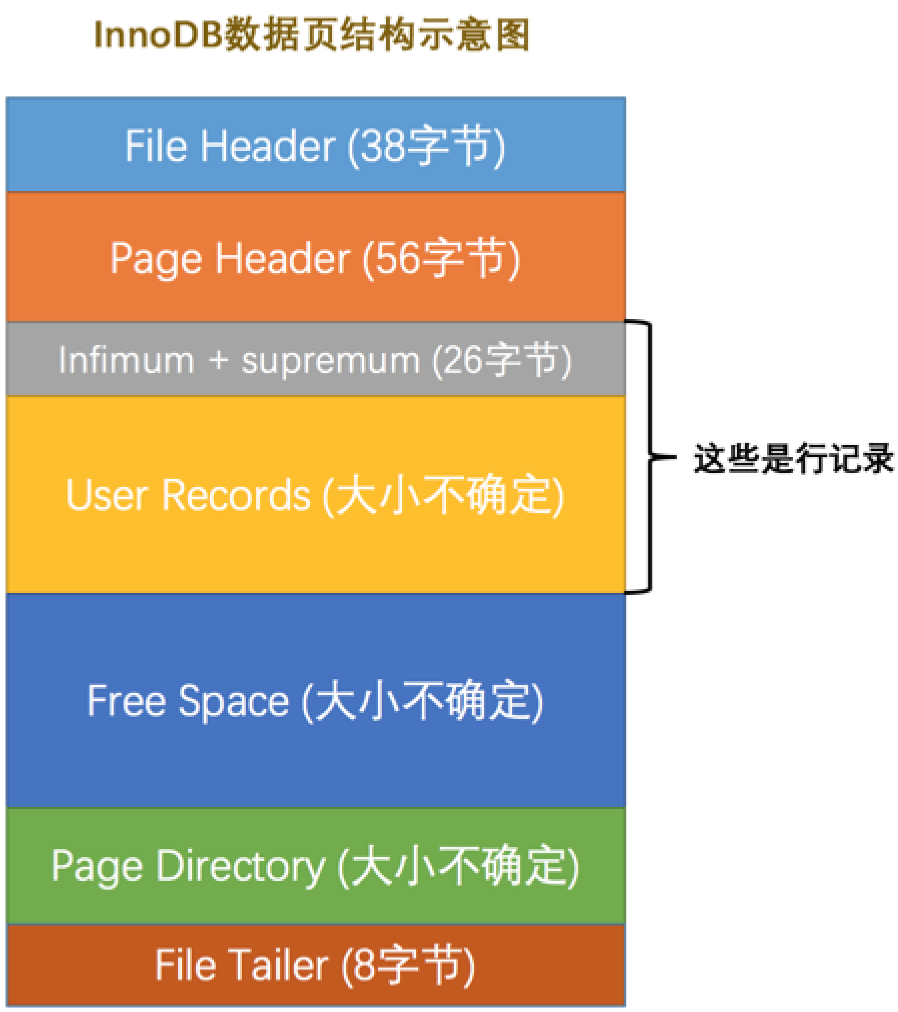
标签： MySQL是怎样运行的

5.1 不同类型的页简介

前边我们简单提了一下 页 的概念，它是 InnoDB 管理存储空间的基本单位，一个页的大小一般是 16KB 。 InnoDB 为了不同的目的而设计了许多种不同类型的 页 ，比如存放表空间头部信息的页，存放 Insert Buffer 信息的页，存放 INODE 信息的页，存放 undo 日志信息的页等等等等。当然了，如果我说的这些名词你一个都没有听过，就当我放了个屁吧～ 不过这没有一毛钱关系，我们今儿个也不准备说这些类型的页，我们聚焦的是那些存放我们表中记录的那种类型的页，官方称这种存放记录的页为索引（ INDEX ）页，鉴于我们还没有了解过索引是个什么东西，而这些表中的记录就是我们日常口中所称的 数据 ，所以目前还是叫这种存放记录的页为 数据页 吧。

5.2 数据页结构的快速浏览

数据页代表的这块 16KB 大小的存储空间可以被划分为多个部分，不同部分有不同的功能，各个部分如图所示：



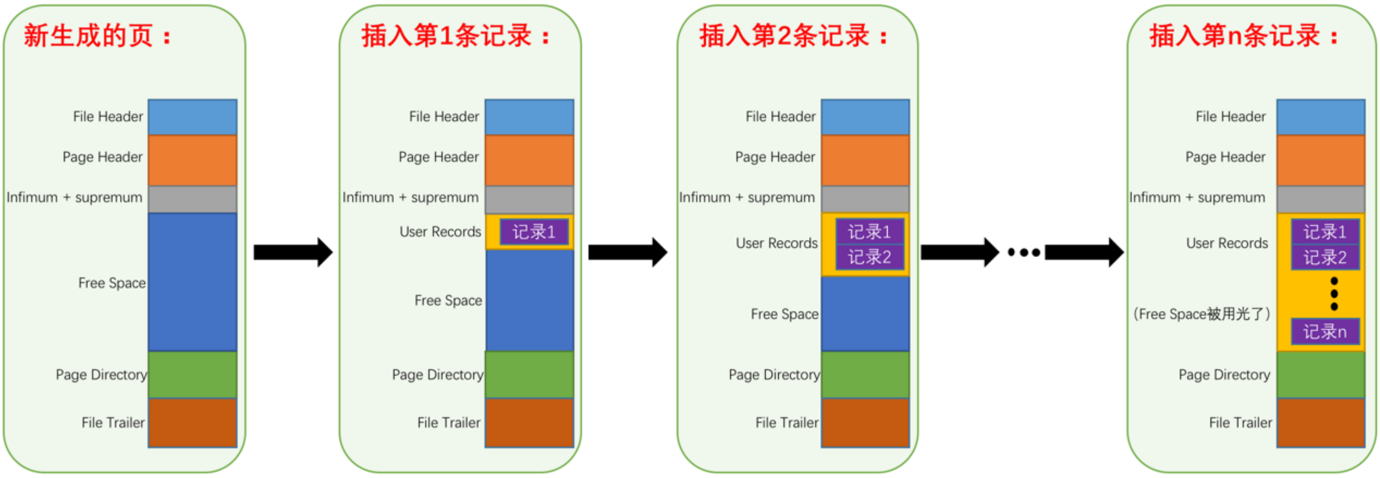
从图中可以看出，一个 InnoDB 数据页的存储空间大致被划分成了 7 个部分，有的部分占用的字节数是确定的，有的部分占用的字节数是不确定的。下边我们用表格的方式来大致描述一下这7个部分都存储一些啥内容（快速的瞅一眼就行了，后边会详细唠叨的）：

名称	中文名	占用空间大小	简单描述
File Header	文件头部	38 字节	页的一些通用信息
Page Header	页面头部	56 字节	数据页专有的一些信息
Infimum + Supremum	最小记录和最大记录	26 字节	两个虚拟的行记录
User Records	用户记录	不确定	实际存储的行记录内容
Free Space	空闲空间	不确定	页中尚未使用的空间
Page Directory	页面目录	不确定	页中的某些记录的相对位置
File Trailer	文件尾部	8 字节	校验页是否完整

小贴士：
我们接下来并不打算按照页中各个部分的出现顺序来依次介绍它们，因为各个部分中会出现很多大家目前不理解的概念，这会打击各位读文章的信心与兴趣，希望各位能接受这种拍摄手法～

5.3 记录在页中的存储

在页的7个组成部分中，我们自己存储的记录会按照我们指定的 行格式 存储到 User Records 部分。但是在一开始生成页的时候，其实并没有 User Records 这个部分，每当我们插入一条记录，都会从 Free Space 部分，也就是尚未使用的存储空间中申请一个记录大小的空间划分到 User Records 部分，当 Free Space 部分的空间全部被 User Records 部分替代掉之后，也就意味着这个页使用完了，如果还有新的记录插入的话，就需要去申请新的页了，这个过程的图示如下：



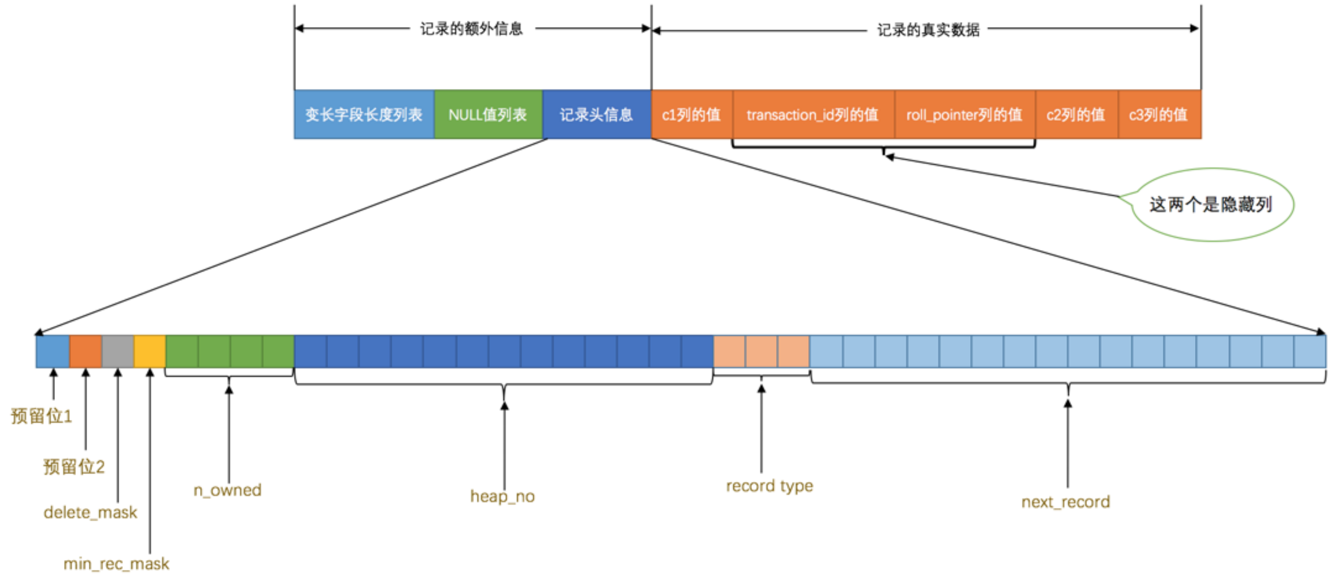
为了更好的管理在 User Records 中的这些记录， InnoDB 可费了一番力气呢，在哪费力气了呢？不就是把记录按照指定的行格式一条一条摆在 User Records 部分么？其实这话还得从记录行格式的 记录头信息 中说起。

5.3.1 记录头信息的秘密

为了故事的顺利发展，我们先创建一个表：

```
mysql> CREATE TABLE page_demo(
    ->     c1 INT,
    ->     c2 INT,
    ->     c3 VARCHAR(10000),
    ->     PRIMARY KEY (c1)
    -> ) CHARSET=ascii ROW_FORMAT=Compact;
Query OK, 0 rows affected (0.03 sec)
```

这个新创建的 page_demo 表有3个列，其中 c1 和 c2 列是用来存储整数的， c3 列是用来存储字符串的。需要注意的是，我们把 ***c1*** 列指定为主键，所以在具体的行格式中InnoDB就没必要为我们去创建那个所谓的 ***row_id*** 隐藏列了。而且我们为这个表指定了 ascii 字符集以及 Compact 的行格式。所以这个表中记录的行格式示意图就是这样的：

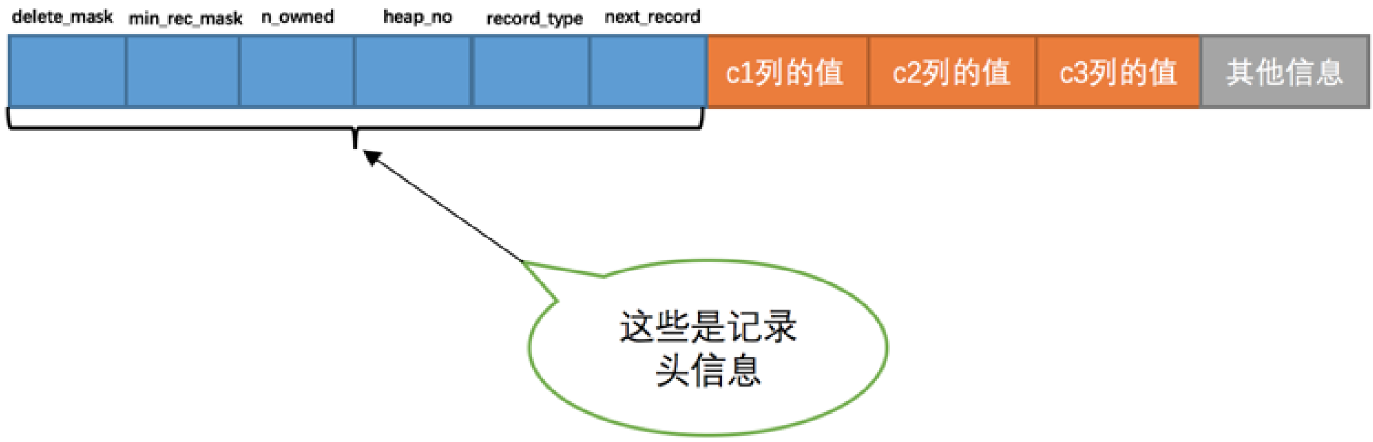


从图中可以看到，我们特意把 记录头信息 的5个字节的数据给标出来了，说明它很重要，我们再次先把这些 记录头信息 中各个属性的大体意思浏览一下（我们目前使用 Compact 行格式进行演示）：

名称	大小（单位：bit）	描述
预留位1	1	没有使用
预留位2	1	没有使用
delete_mask	1	标记该记录是否被删除
min_rec_mask	1	B+树的每层非叶子节点中的最小记录都会添加该标记
n_owned	4	表示当前记录拥有的记录数
heap_no	13	表示当前记录在记录堆的位置信息
record_type	3	表示当前记录的类型， 0 表示普通记录， 1 表示B+树非叶节点记录， 2 表示最小记录， 3 表示最大记录
next_record	16	表示下一条记录的相对位置

由于我们现在主要在唠叨 记录头信息 的作用，所以为了大家理解上的方便，我们只在 page_demo 表的行格式演示图中画出有关的头信息属性以及 c1 、 c2 、 c3 列的信息（其他信息没画不代表它们不存在啊，只是为了理解上的方便在图中省略了～），简化后的行格式示意图就是这样：

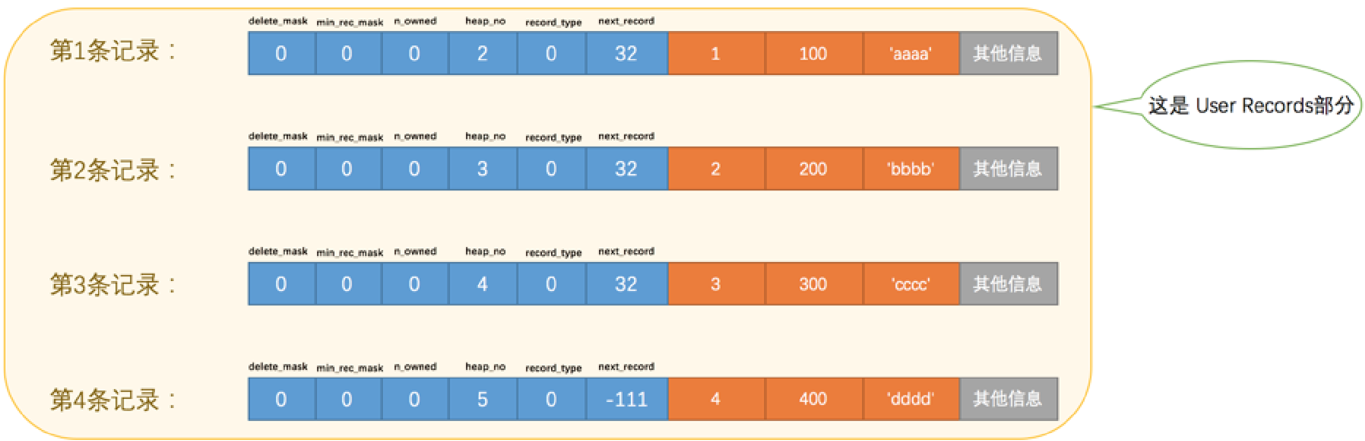
page_demo表的行格式简化图



下边我们试着向 page_demo 表中插入几条记录：

```
mysql> INSERT INTO page_demo VALUES(1, 100, 'aaaa'), (2, 200, 'bbbb'), (3, 300, 'cccc'),
 (4, 400, 'dddd');
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

为了方便大家分析这些记录在 页 的 User Records 部分中是怎么表示的，我把记录中头信息和实际的列数据都用十进制表示出来了（其实是一堆二进制位），所以这些记录的示意图就是：



看这个图的时候需要注意一下，各条记录在 User Records 中存储的时候并没有空隙，这里只是为了大家观看方便才把每条记录单独画在一行中。我们对照着这个图来看看记录头信息中的各个属性是啥意思：

- delete_mask

  这个属性标记着当前记录是否被删除，占用1个二进制位，值为 0 的时候代表记录并没有被删除，为 1 的时候代表记录被删除掉了。

  啥？被删除的记录还在 页 中么？是的，摆在台面上的和背地里做的可能大相径庭，你以为它删除了，可它还在真实的磁盘上[摊手]（忽然想起冠希～）。这些被删除的记录之所以不立即从磁盘上移除，是因为移除它们之后把其他的记录在磁盘上重新排列需要性能消耗，所以只是打一个删除标记而已，所有被删除掉的记录都会组成一个所谓的 垃圾链表 ，在这个链表中的记录占用的空间称之为所谓的 可重用空间 ，之后如果有新记录插入到表中的话，可能把这些被删除的记录占用的存储空间覆盖掉。

> 小贴士：
> 将这个delete_mask位设置为1和将被删除的记录加入到垃圾链表中其实是两个阶段，我们后边在介绍事务的时候会详细唠叨删除操作的详细过程，稍安勿躁。

- min_rec_mask

  B+树的每层非叶子节点中的最小记录都会添加该标记，什么是个 B+ 树？什么是个非叶子节点？好吧，等会再聊这个问题。反正我们自己插入的四条记录的 min_rec_mask 值都是 0 ，意味着它们都不是 B+ 树的非叶子节点中的最小记录。
- n_owned

  这个暂时保密，稍后它是主角~
- heap_no

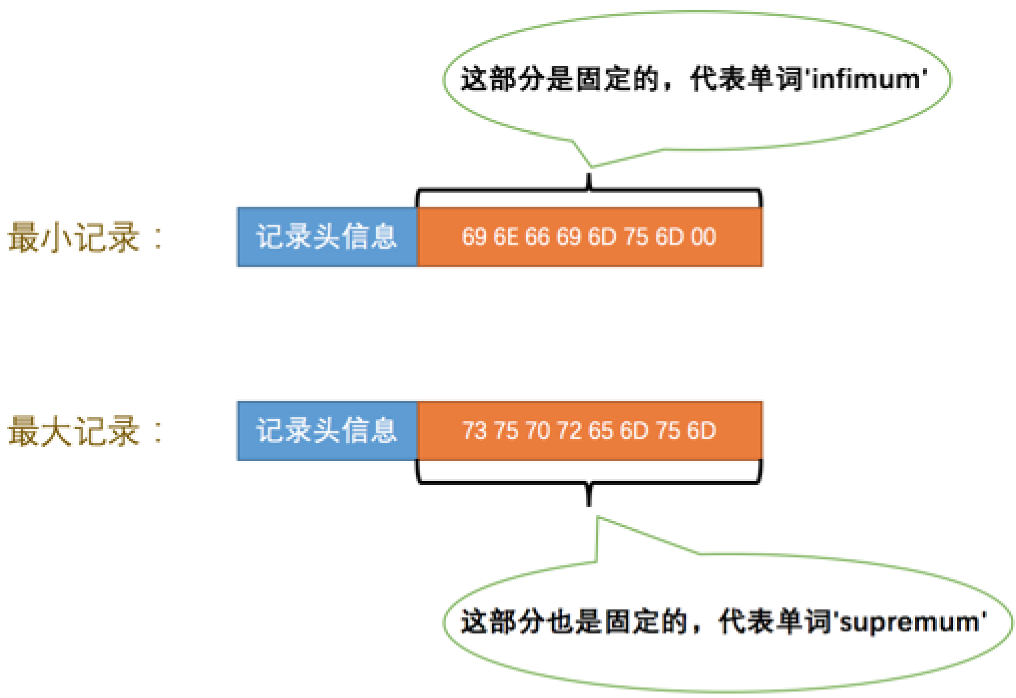
  这个属性表示当前记录在本 页 中的位置，从图中可以看出来，我们插入的4条记录在本 页 中的位置分别是： 2 、 3 、 4 、 5 。是不是少了点啥？是的，怎么不见 heap_no 值为 0 和 1 的记录呢？

  这其实是设计 InnoDB 的大叔们玩的一个小把戏，他们自动给每个页里边儿加了两个记录，由于这两个记录并不是我们自己插入的，所以有时候也称为 伪记录 或者 虚拟记录 。这两个伪记录一个代表 最小记录 ，一个代表 最大记录 ，等一下哈~，记录可以比大小么？

  是的，记录也可以比大小，对于一条完整的记录来说，比较记录的大小就是比较 主键 的大小。比方说我们插入的4行记录的主键值分别是： 1 、 2 、 3 、 4 ，这也就意味着这4条记录的大小从小到大依次递增。

  > 小贴士：
  > 请注意我强调了对于`一条完整的记录`来说，比较记录的大小就相当于比的是主键的大小。后边我们还会介绍只存储一条记录的部分列的情况，敬请期待~

  但是不管我们向 页 中插入了多少自己的记录，设计 InnoDB 的大叔们都规定他们定义的两条伪记录分别为最小记录与最大记录。这两条记录的构造十分简单，都是由5字节大小的 记录头信息 和8字节大小的一个固定的部分组成的，如图所示



  由于这两条记录不是我们自己定义的记录，所以它们并不存放在 页 的 User Records 部分，他们被单独放在一个称为 Infimum + Supremum 的部分，如图所示：



从图中我们可以看出来，最小记录和最大记录的 heap_no 值分别是 0 和 1 ，也就是说它们的位置最靠前。

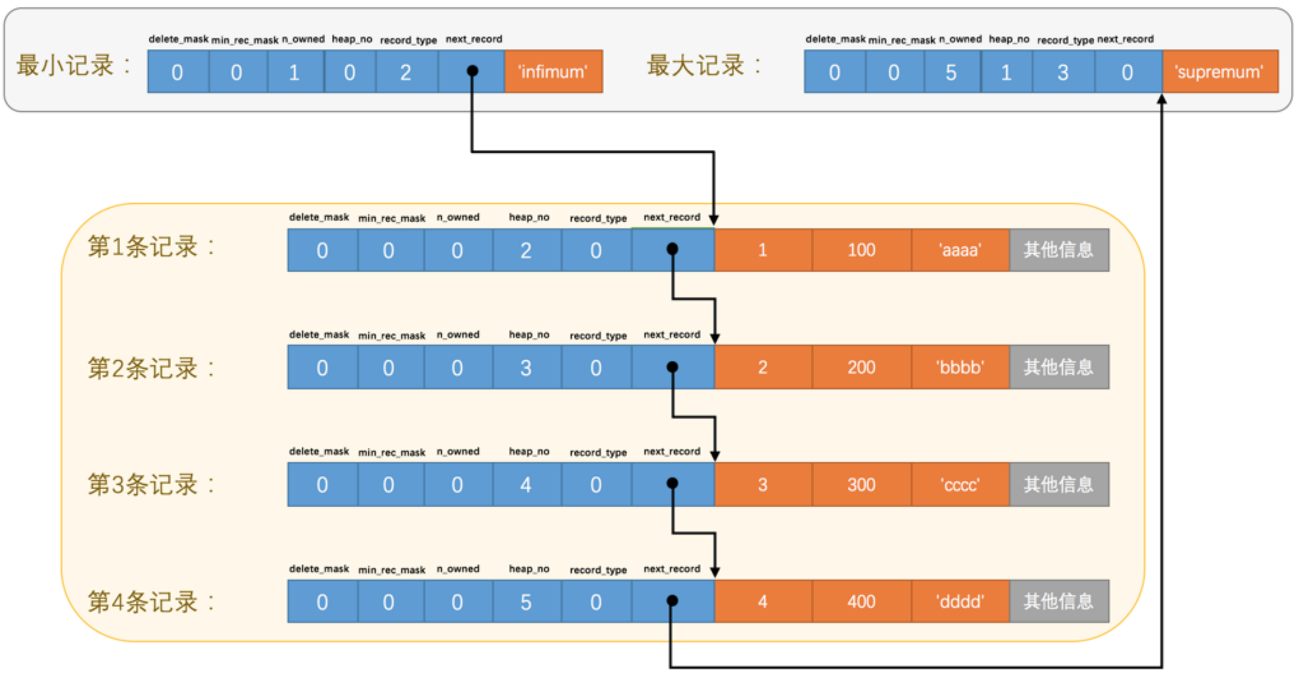
- record_type

这个属性表示当前记录的类型，一共有4种类型的记录， 0 表示普通记录， 1 表示B+树非叶节点记录， 2 表示最小记录， 3 表示最大记录。从图中我们也可以看出来，我们自己插入的记录就是普通记录，它们的 record_type 值都是 0 ，而最小记录和最大记录的 record_type 值分别为 2 和 3 。

至于 record_type 为 1 的情况，我们之后在说索引的时候会重点强调的。

- next_record

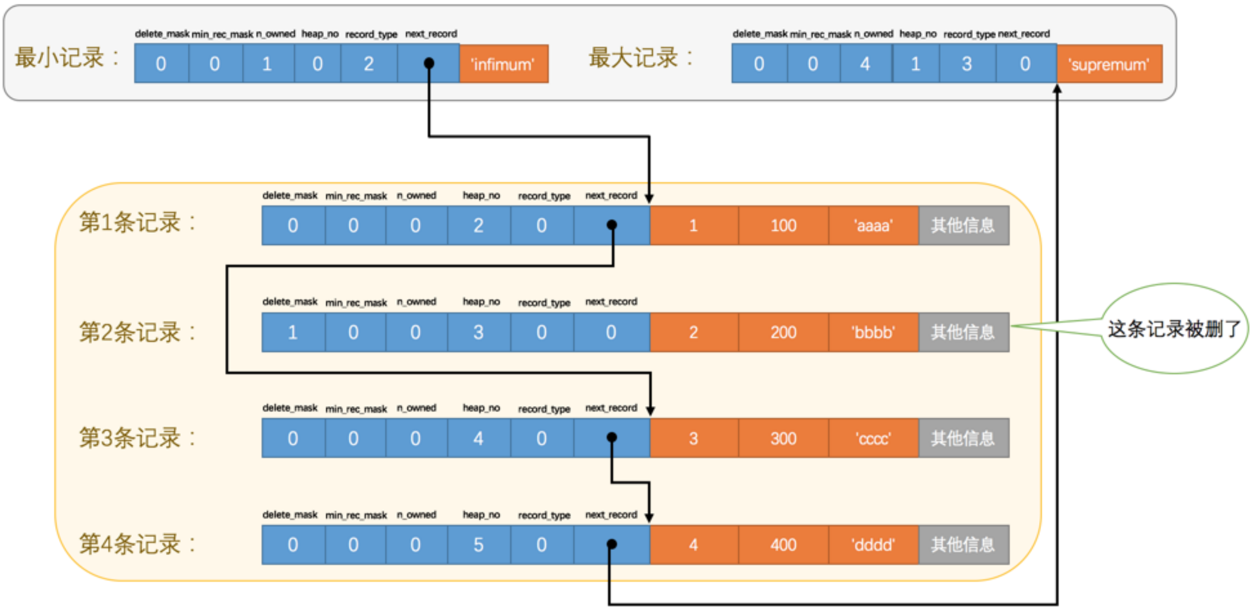
这玩意儿非常重要，它表示从当前记录的真实数据到下一条记录的真实数据的地址偏移量。比方说第一条记录的 next_record 值为 32 ，意味着从第一条记录的真实数据的地址处向后找 32 个字节便是下一条记录的真实数据。如果你熟悉数据结构的话，就立即明白了，这其实是个 链表 ，可以通过一条记录找到它的下一条记录。但是需要注意注意再注意的一点是， 下一条记录 指得并不是按照我们插入顺序的下一条记录，而是按照主键值由小到大的顺序的下一条记录。而且规定 ***Infimum记录（也就是最小记录）*** 的下一条记录就是本页中主键值最小的用户记录，而本页中主键值最大的用户记录的下一条记录就是 ***Supremum记录（也就是最大记录）*** ，为了更形象的表示一下这个 next_record 起到的作用，我们用箭头来替代一下 next_record 中的地址偏移量：



从图中可以看出来，我们的记录按照主键从小到大的顺序形成了一个单链表。 最大记录 的 next_record 的值为 0 ，这也就是说最大记录是没有 下一条记录 了，它是这个单链表中的最后一个节点。如果从中删除掉一条记录，这个链表也是会跟着变化的，比如我们把第2条记录删掉：

```
mysql> DELETE FROM page_demo WHERE c1 = 2;
Query OK, 1 row affected (0.02 sec)
```

删掉第2条记录后的示意图就是：



从图中可以看出来，删除第2条记录前后主要发生了这些变化：

- 第2条记录并没有从存储空间中移除，而是把该条记录的 delete_mask 值设置为 1 。
- 第2条记录的 next_record 值变为了0，意味着该记录没有下一条记录了。
- 第1条记录的 next_record 指向了第3条记录。
- 还有一点你可能忽略了，就是 最大记录 的 n_owned 值从 5 变成了 4 ，关于这一点的变化我们稍后会详细说明的。

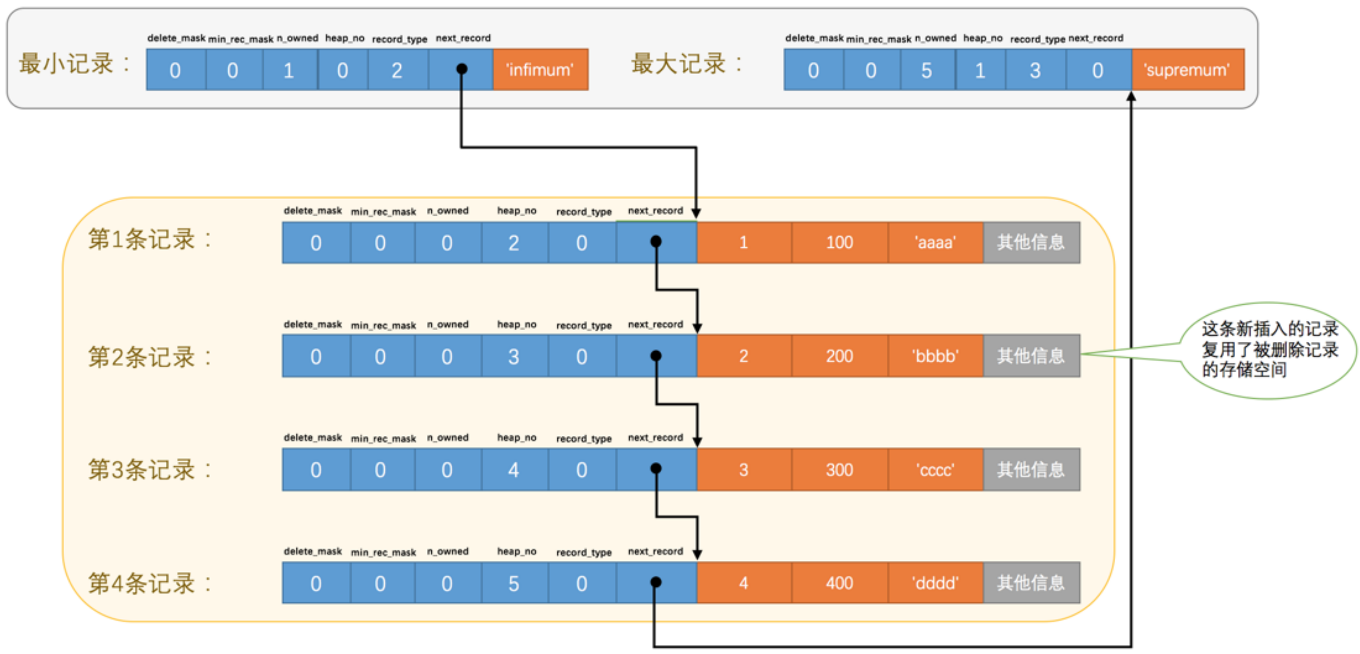
  所以，不论我们怎么对页中的记录做增删改操作，InnoDB始终会维护一条记录的单链表，链表中的各个节点是按照主键值由小到大的顺序连接起来的。

> 小贴士：
> 你会不会觉得next_record这个指针有点儿怪，为啥要指向记录头信息和真实数据之间的位置呢？为啥不干脆指向整条记录的开头位置，也就是记录的额外信息开头的位置呢？
> 因为这个位置刚刚好，向左读取就是记录头信息，向右读取就是真实数据。我们前边还说过变长字段长度列表、NULL值列表中的信息都是逆序存放，这样可以使记录中位置靠前的字段和它们对应的字段长度信息在内存中的距离更近，可能会提高高速缓存的命中率。当然如果你看不懂这句话的话就不要勉强了，果断跳过～

再来看一个有意思的事情，因为主键值为 2 的记录被我们删掉了，但是存储空间却没有回收，如果我们再次把这条记录插入到表中，会发生什么事呢？

```
mysql> INSERT INTO page_demo VALUES(2, 200, 'bbbb');
Query OK, 1 row affected (0.00 sec)
```

我们看一下记录的存储情况：



从图中可以看到， InnoDB 并没有因为新记录的插入而为它申请新的存储空间，而是直接复用了原来被删除记录的存储空间。

> 小贴士：
> 当数据页中存在多条被删除掉的记录时，这些记录的next_record属性将会把这些被删除掉的记录组成一个垃圾链表，以备之后重用这部分存储空间。

5.4 Page Directory（页目录）

现在我们了解了记录在页中按照主键值由小到大顺序串联成一个单链表，那如果我们想根据主键值查找页中的某条记录该咋办呢？比如说这样的查询语句：

```
SELECT * FROM page_demo WHERE c1 = 3;
```

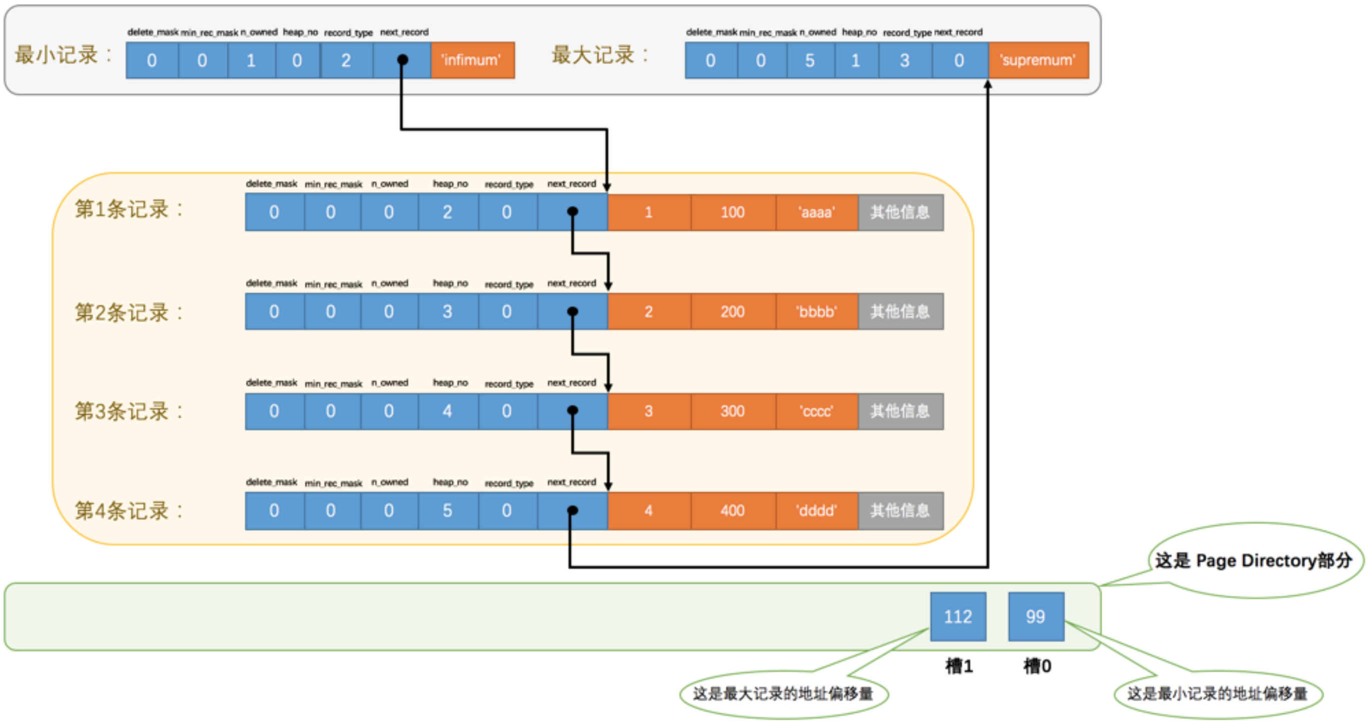
最笨的办法：从 Infimum 记录（最小记录）开始，沿着链表一直往后找，总有一天会找到（或者找不到[摊手]），在找的时候还能投机取巧，因为链表中各个记录的值是按照从小到大顺序排列的，所以当链表的某个节点代表的记录的主键值大于你想要查找的主键值时，你就可以停止查找了，因为该节点后边的节点的主键值依次递增。

这个方法在页中存储的记录数量比较少的情况用起来也没啥问题，比方说现在我们的表里只有 4 条自己插入的记录，所以最多找 4 次就可以把所有记录都遍历一遍，但是如果一个页中存储了非常多的记录，这么查找对性能来说还是有损耗的，所以我们说这种遍历查找这是一个 笨 办法。但是设计 InnoDB 的大叔们是什么人，他们能用这么笨的办法么，当然是要设计一种更6的查找方式喽，他们从书的目录中找到了灵感。

我们平常想从一本书中查找某个内容的时候，一般会先看目录，找到需要查找的内容对应的书的页码，然后到对应的页码查看内容。设计 InnoDB 的大叔们为我们的记录也制作了一个类似的目录，他们的制作过程是这样的：

1. 将所有正常的记录（包括最大和最小记录，不包括标记为已删除的记录）划分为几个组。
2. 每个组的最后一条记录（也就是组内最大的那条记录）的头信息中的 n_owned 属性表示该记录拥有多少条记录，也就是该组内共有几条记录。
3. 将每个组的最后一条记录的地址偏移量单独提取出来按顺序存储到靠近 页 的尾部的地方，这个地方就是所谓的 Page Directory ，也就是 页目录 （此时应该返回头看看页面各个部分的图）。页面目录中的这些地址偏移量被称为 槽 （英文名： Slot ），所以这个页面目录就是由 槽 组成的。

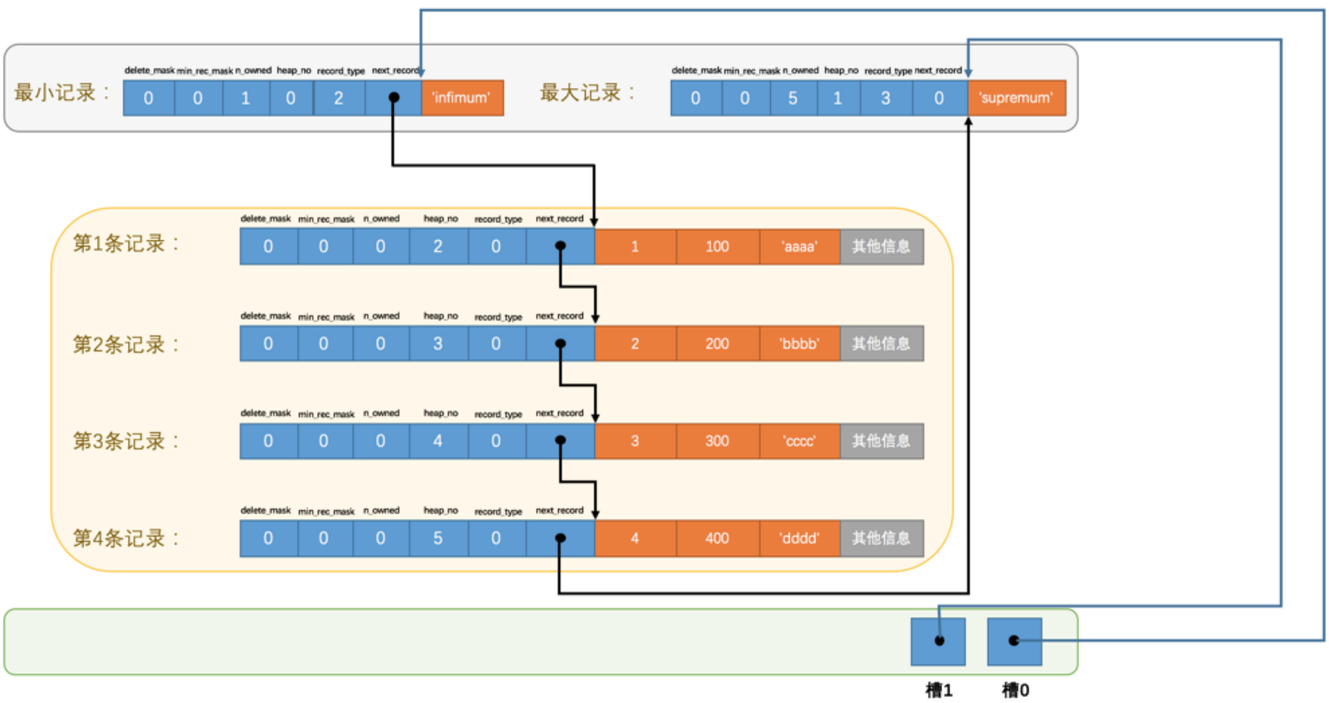
比方说现在的 page_demo 表中正常的记录共有6条， InnoDB 会把它们分成两组，第一组中只有一个最小记录，第二组中是剩余的5条记录，看下边的示意图：



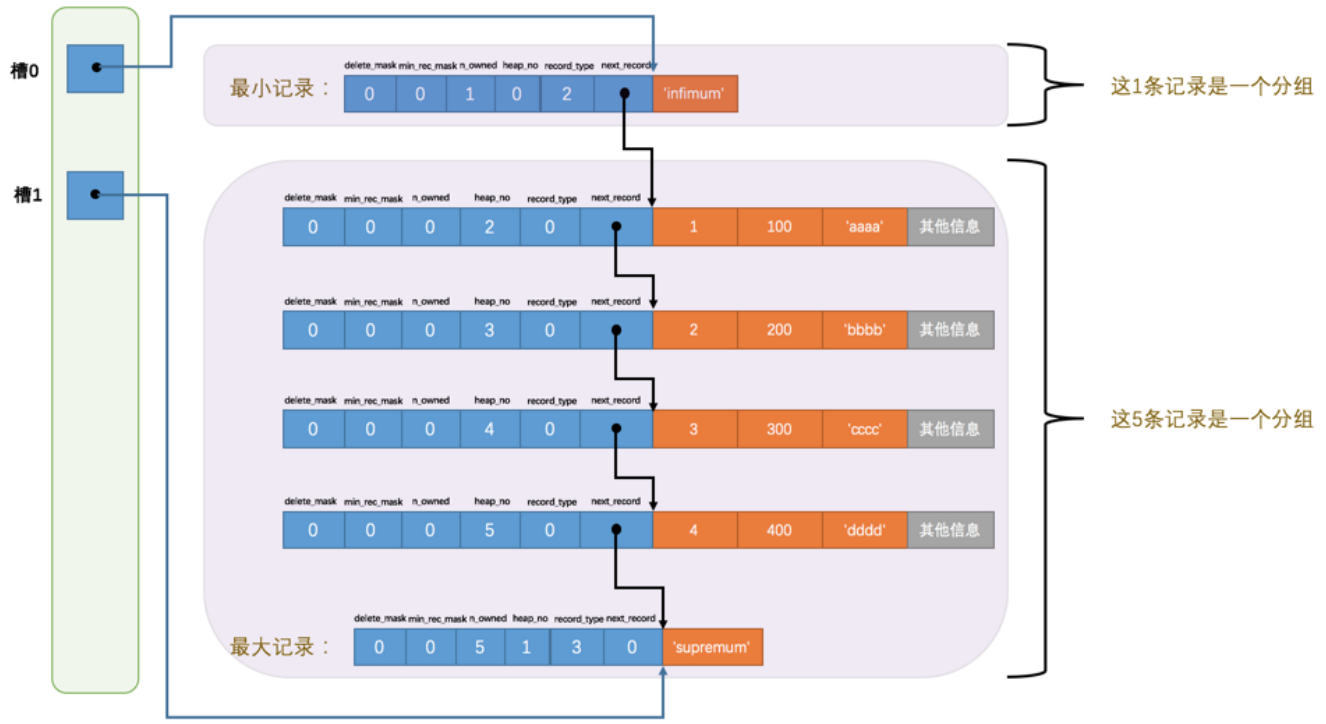
从这个图中我们需要注意这么几点：

- 现在 页目录 部分中有两个槽，也就意味着我们的记录被分成了两个组， 槽1 中的值是 112 ，代表最大记录的地址偏移量（就是从页面的0字节开始数，数112个字节）； 槽0 中的值是 99 ，代表最小记录的地址偏移量。
- 注意最小和最大记录的头信息中的 n_owned 属性
  - 最小记录的 n_owned 值为 1 ，这就代表着以最小记录结尾的这个分组中只有 1 条记录，也就是最小记录本身。
  - 最大记录的 n_owned 值为 5 ，这就代表着以最大记录结尾的这个分组中只有 5 条记录，包括最大记录本身还有我们自己插入的 4 条记录。

99 和 112 这样的地址偏移量很不直观，我们用箭头指向的方式替代数字，这样更易于我们理解，所以修改后的示意图就是这样：



哎呀，咋看上去怪怪的，这么乱的图对于我这个强迫症真是不能忍，那我们就暂时不管各条记录在存储设备上的排列方式了，单纯从逻辑上看一下这些记录和页目录的关系：



这样看就顺眼多了嘛！为什么最小记录的 n_owned 值为1，而最大记录的 n_owned 值为 5 呢，这里头有什么猫腻么？

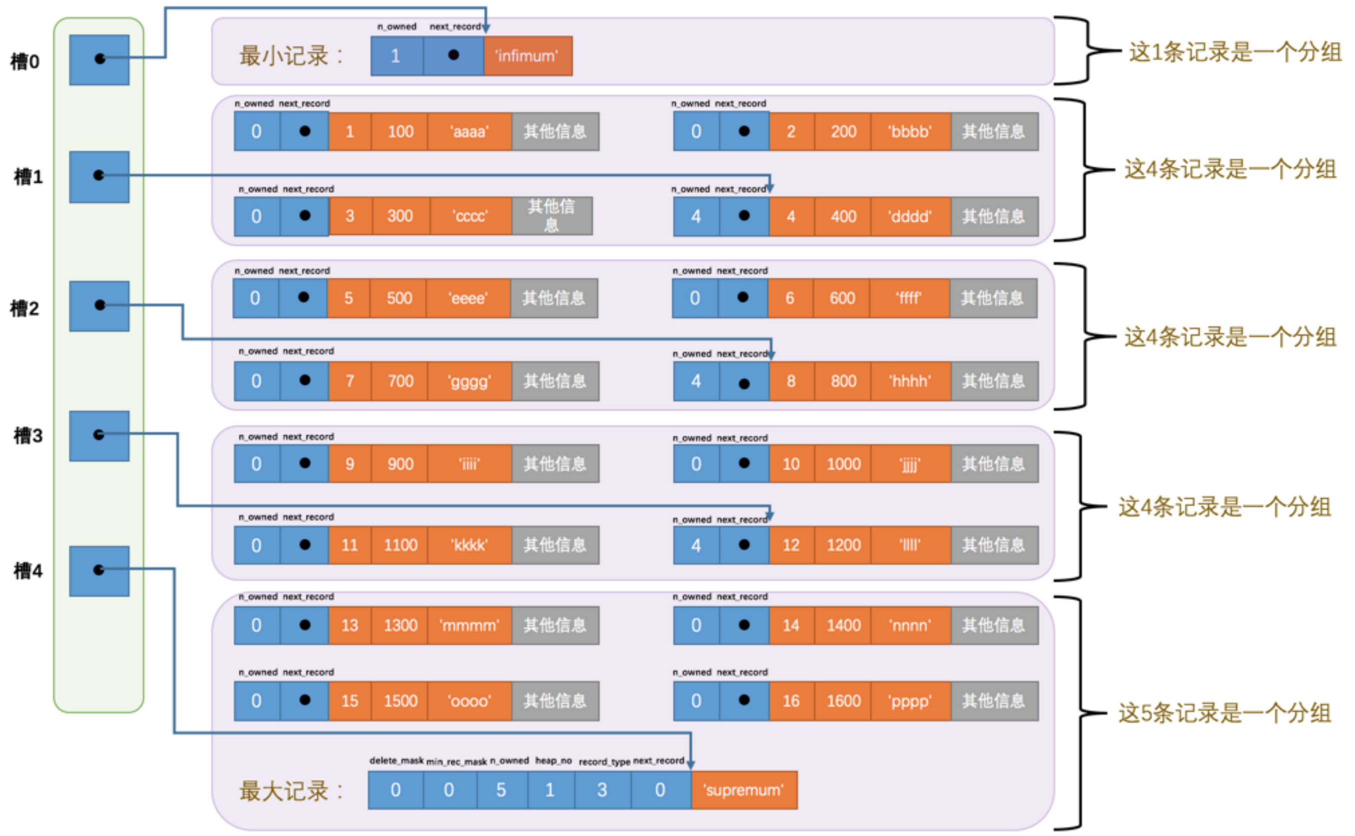
是的，设计 InnoDB 的大叔们对每个分组中的记录条数是有规定的：**对于最小记录所在的分组只能有 *1* 条记录，最大记录所在的分组拥有的记录条数只能在 *1~8* 条之间，剩下的分组中记录的条数范围只能在是 *4~8* 条之间。**所以分组是按照下边的步骤进行的：

- 初始情况下一个数据页里只有最小记录和最大记录两条记录，它们分属于两个分组。
- 之后每插入一条记录，都会从 页目录 中找到主键值比本记录的主键值大并且差值最小的槽，然后把该槽对应的记录的 n_owned 值加1，表示本组内又添加了一条记录，直到该组中的记录数等于8个。
- 在一个组中的记录数等于8个后再插入一条记录时，会将组中的记录拆分成两个组，一个组中4条记录，另一个5条记录。这个过程会在 页目录 中新增一个 槽 来记录这个新增分组中最大的那条记录的偏移量。

由于现在 page_demo 表中的记录太少，无法演示添加了 页目录 之后加快查找速度的过程，所以再往 page_demo 表中添加一些记录：

```
mysql> INSERT INTO page_demo VALUES(5, 500, 'eeee'), (6, 600, 'ffff'), (7, 700, 'gggg'),
 (8, 800, 'hhhh'), (9, 900, 'iiii'), (10, 1000, 'jjjj'), (11, 1100, 'kkkk'), (12, 1200, 'l
lll'), (13, 1300, 'mmmm'), (14, 1400, 'nnnn'), (15, 1500, 'oooo'), (16, 1600, 'pppp');
Query OK, 12 rows affected (0.00 sec)
Records: 12  Duplicates: 0  Warnings: 0
```

哈，我们一口气又往表中添加了12条记录，现在页里边就一共有18条记录了（包括最小和最大记录），这些记录被分成了5个组，如图所示：



因为把16条记录的全部信息都画在一张图里太占地方，让人眼花缭乱的，所以只保留了用户记录头信息中的 n_owned 和 next_record 属性，也省略了各个记录之间的箭头，我没画不等于没有啊！现在看怎么从这个 页目录 中查找记录。因为各个槽代表的记录的主键值都是从小到大排序的，所以我们可以使用所谓的 二分法 来进行快速查找。4个槽的编号分别是： 0 、 1 、 2 、 3 、 4 ，所以初始情况下最低的槽就是 low=0 ，最高的槽就是 high=4 。比方说我们想找主键值为 6 的记录，过程是这样的：

1. 计算中间槽的位置： (0+4)/2=2 ，所以查看 槽2 对应记录的主键值为 8 ，又因为 8 > 6 ，所以设置 high=2 ， low 保持不变。
2. 重新计算中间槽的位置： (0+2)/2=1 ，所以查看 槽1 对应的主键值为 4 ，又因为 4 < 6 ，所以设置 low=1 ， high 保持不变。
3. 因为 high - low 的值为1，所以确定主键值为 5 的记录在 槽2 对应的组中。此刻我们需要找到 槽2 中主键值最小的那条记录，然后沿着单向链表遍历 槽2 中的记录。但是我们前边又说过，每个槽对应的记录都是该组中主键值最大的记录，这里 槽2 对应的记录是主键值为 8 的记录，怎么定位一个组中最小的记录呢？别忘了各个槽都是挨着的，我们可以很轻易的拿到 槽1 对应的记录（主键值为 4 ），该条记录的下一条记录就是 槽2 中主键值最小的记录，该记录的主键值为 5 。所以我们可以从这条主键值为 5 的记录出发，遍历 槽2 中的各条记录，直到找到主键值为 6 的那条记录即可。由于一个组中包含的记录条数只能是1~8条，所以遍历一个组中的记录的代价是很小的。

所以在一个数据页中查找指定主键值的记录的过程分为两步：

1. 通过二分法确定该记录所在的槽，并找到该槽中主键值最小的那条记录。
2. 通过记录的 next_record 属性遍历该槽所在的组中的各个记录。

> 小贴士：
> 如果你不知道二分法是个什么东西，找个基础算法书看看吧。什么？算法书写的看不懂？等我～

5.5 Page Header（页面头部）

设计 InnoDB 的大叔们为了能得到一个数据页中存储的记录的状态信息，比如本页中已经存储了多少条记录，第一条记录的地址是什么，页目录中存储了多少个槽等等，特意在页中定义了一个叫 Page Header 的部分，它是 页 结构的第二部分，这个部分占用固定的 56 个字节，专门存储各种状态信息，具体各个字节都是干嘛的看下

表:

名称	占用空间大小	描述
PAGE_N_DIR_SLOTS	2 字节	在页目录中的槽数量
PAGE_HEAP_TOP	2 字节	还未使用的空间最小地址，也就是说从该地址之后就是 Free Space
PAGE_N_HEAP	2 字节	本页中的记录的数量（包括最小和最大记录以及标记为删除的记录）
PAGE_FREE	2 字节	第一个已经标记为删除的记录地址（各个已删除的记录通过 next_record 也会组成一个单链表，这个单链表中的记录可以被重新利用）
PAGE_GARBAGE	2 字节	已删除记录占用的字节数
PAGE_LAST_INSERT	2 字节	最后插入记录的位置
PAGE_DIRECTION	2 字节	记录插入的方向
PAGE_N_DIRECTION	2 字节	一个方向连续插入的记录数量
PAGE_N_RECS	2 字节	该页中记录的数量（不包括最小和最大记录以及被标记为删除的记录）
PAGE_MAX_TRX_ID	8 字节	修改当前页的最大事务ID，该值仅在二级索引中定义
PAGE_LEVEL	2 字节	当前页在B+树中所处的层级
PAGE_INDEX_ID	8 字节	索引ID，表示当前页属于哪个索引
PAGE_BTR_SEG_LEAF	10 字节	B+树叶子段的头部信息，仅在B+树的Root页定义
PAGE_BTR_SEG_TOP	10 字节	B+树非叶子段的头部信息，仅在B+树的Root页定义

如果大家认真看过前边的文章，从 PAGE_N_DIR_SLOTS 到 PAGE_LAST_INSERT 以及 PAGE_N_RECS 的意思大家一定是清楚的，如果不清楚，对不起，你应该回头再看一遍前边的文章。剩下的状态信息看不明白不要着急，饭要一口一口吃，东西要一点一点学（一定要稍安勿躁哦，不要被这些名词吓到）。在这里我们先唠叨一下 PAGE_DIRECTION 和 PAGE_N_DIRECTION 的意思:

- PAGE_DIRECTION

  假如新插入的一条记录的主键值比上一条记录的主键值大，我们说这条记录的插入方向是右边，反之则是左边。用来表示最后一条记录插入方向的状态就是 PAGE_DIRECTION 。
- PAGE_N_DIRECTION

  假设连续几次插入新记录的方向都是一致的， InnoDB 会把沿着同一个方向插入记录的条数记下来，这个条数就用 PAGE_N_DIRECTION 这个状态表示。当然，如果最后一条记录的插入方向改变了的话，这个状态的值会被清零重新统计。

至于我们没提到的那些属性，我没说是因为现在不需要大家知道。不要着急，当我们学完了后边的内容，你再回头看，一切都是那么清晰。

> 小贴士：
> 说到这个有些东西后边我们学过后回头看就很清晰的事儿不禁让我想到了乔布斯在斯坦福大学的演讲，摆一下原文：
> “You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future.You have to trust in something - your gut, destiny, life, karma, whatever. This approach has never let me down, and it has made all the difference in my life.”
> 上边这段话纯属心血来潮写的，大意是坚持做自己喜欢的事儿，你在做的时候可能并不能搞清楚这些事儿对自己之后的人生有啥影响，但当你一路走来回头看时，一切都是那么清晰，就像是命中注定的一样。上述内容跟MySQL毫无干系，请忽略～

5.6 File Header（文件头部）

上边唠叨的 Page Header 是专门针对 数据页 记录的各种状态信息，比方说页里头有多少个记录了呀，有多少个槽了呀。我们现在描述的 File Header 针对各种类型的页都通用，也就是说不同类型的页都会以 File Header 作为第一个组成部分，它描述了一些针对各种页都通用的一些信息，比方说这个页的编号是多少，它的上一个页、下一个页是谁啦吧啦吧啦~ 这个部分占用固定的 38 个字节，是由下边这些内容组成的：

名称	占用空间大小	描述
FIL_PAGE_SPACE_OR_CHKSUM	4 字节	页的校验和（checksum值）
FIL_PAGE_OFFSET	4 字节	页号
FIL_PAGE_PREV	4 字节	上一个页的页号
FIL_PAGE_NEXT	4 字节	下一个页的页号
FIL_PAGE_LSN	8 字节	页面被最后修改时对应的日志序列位置（英文名是：Log Sequence Number）
FIL_PAGE_TYPE	2 字节	该页的类型
FIL_PAGE_FILE_FLUSH_LSN	8 字节	仅在系统表空间的一个页中定义，代表文件至少被刷新到了对应的LSN值
FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID	4 字节	页属于哪个表空间

对照着这个表格，我们看几个目前比较重要的部分：

- FIL_PAGE_SPACE_OR_CHKSUM

  这个代表当前页面的校验和（checksum）。啥是个校验和？就是对于一个很长很长的字节串来说，我们会通过某种算法来计算一个比较短的值来代表这个很长的字节串，这个比较短的值就称为 校验和 。这样在比较两个很长的字节串之前先比较这两个长字节串的校验和，如果校验和都不一样两个长字节串肯定是不同的，所以省去了直接比较两个比较长的字节串的时间损耗。
- FIL_PAGE_OFFSET

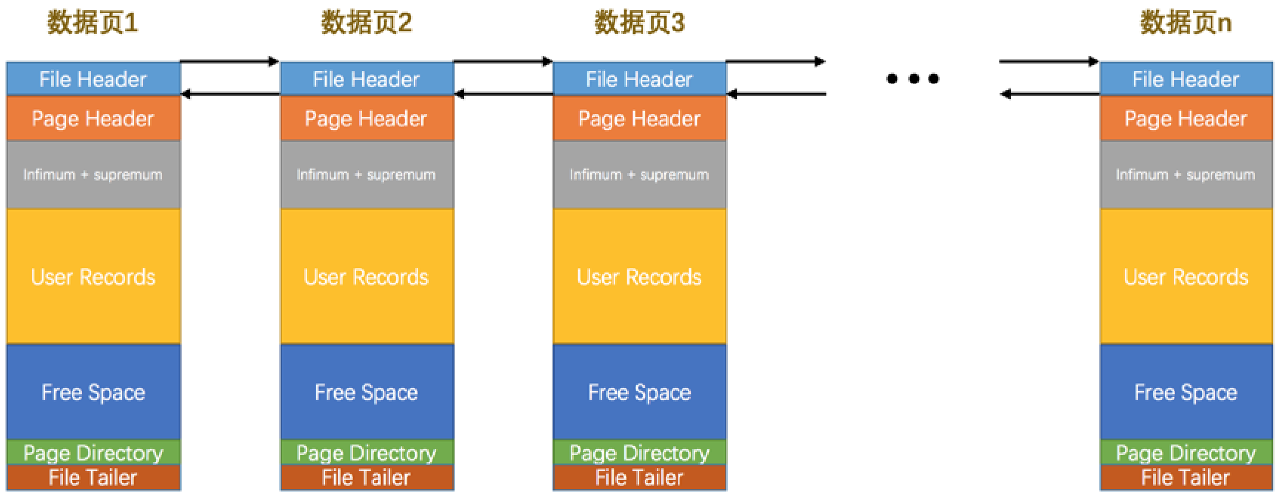
  每一个 页 都有一个单独的页号，就跟你的身份证号码一样， InnoDB 通过页号来可以唯一定位一个 页 。
- FIL_PAGE_TYPE

  这个代表当前 页 的类型，我们前边说过， InnoDB 为了不同的目的而把页分为不同的类型，我们上边介绍的其实都是存储记录的 数据页 ，其实还有很多别的类型的页，具体如下表： |类型名称|十六进制|描述| |:--:|:--:|:--:| | FIL_PAGE_TYPE_ALLOCATED |0x0000|最新分配，还没使用| | FIL_PAGE_UNDO_LOG |0x0002|Undo日志页| | FIL_PAGE_INODE |0x0003|段信息节点| | FIL_PAGE_IBUF_FREE_LIST |0x0004|Insert Buffer空闲列表| | FIL_PAGE_IBUF_BITMAP |0x0005|Insert Buffer位图| | FIL_PAGE_TYPE_SYS |0x0006|系统页| | FIL_PAGE_TYPE_TRX_SYS |0x0007|事务系统数据| | FIL_PAGE_TYPE_FSP_HDR |0x0008|表空间头部信息| | FIL_PAGE_TYPE_XDES |0x0009|扩展描述页| | FIL_PAGE_TYPE_BLOB |0x000A|BLOB页| | FIL_PAGE_INDEX |0x45BF|索引页，也就是我们所说的 数据页 |

  我们存放记录的数据页的类型其实是 FIL_PAGE_INDEX ，也就是所谓的 索引页 。至于啥是个索引，且听下回分解~
- FIL_PAGE_PREV 和 FIL_PAGE_NEXT

  我们前边强调过， InnoDB 都是以页为单位存放数据的，有时候我们存放某种类型的数据占用的空间非常大（比方说一张表中可以有成千上万条记录）， InnoDB 可能不可以一次性为这么多数据分配一个非常大的存储空间，如果分散到多个不连续的页中存储的话需要把这些页关联起来， FIL_PAGE_PREV 和 FIL_PAGE_NEXT 就分别代表本页的上一个和下一个页的页号。这样通过建立一个双向链表把许许多多的页就都串联起来了，

而无需这些页在物理上真正连着。需要注意的是，并不是所有类型的页都有上一个和下一个页的属性，不过我们本集中唠叨的 数据页 （也就是类型为 FIL_PAGE_INDEX 的页）是有这两个属性的，所以所有的数据页其实是一个双链表，就像这样：



关于 File Header 的其他属性我们暂时用不到，等用到的时候再提哈～

5.7 File Trailer

我们知道 InnoDB 存储引擎会把数据存储到磁盘上，但是磁盘速度太慢，需要以 页 为单位把数据加载到内存中处理，如果该页中的数据在内存中被修改了，那么在修改后的某个时间需要把数据同步到磁盘中。但是在同步了一半的时候中断电了咋办，这不是莫名尴尬么？为了检测一个页是否完整（也就是在同步的时候有没有发生只同步一半的尴尬情况），设计 InnoDB 的大叔们在每个页的尾部都加了一个 File Trailer 部分，这个部分由 8 个字节组成，可以分成2个小部分：

- 前4个字节代表页的校验和

  这个部分是和 File Header 中的校验和相对应的。每当一个页面在内存中修改了，在同步之前就要把它的校验和算出来，因为 File Header 在页面的前边，所以校验和会被首先同步到磁盘，当完全写完时，校验和也会被写到页的尾部，如果完全同步成功，则页的首部和尾部的校验和应该是一致的。如果写了一半儿断电了，那么在 File Header 中的校验和就代表着已经修改过的页，而在 File Trialer 中的校验和代表着原先的页，二者不同则意味着同步中间出了错。
- 后4个字节代表页面被最后修改时对应的日志序列位置（LSN）

  这个部分也是为了校验页的完整性的，只不过我们目前还没说 LSN 是个什么意思，所以大家可以先不用管这个属性。

这个 File Trailer 与 File Header 类似，都是所有类型的页通用的。

5.8 总结

1. InnoDB为了不同的目的而设计了不同类型的页，我们把用于存放记录的页叫做 数据页 。
2. 一个数据页可以被大致划分为7个部分，分别是

   - File Header ，表示页的一些通用信息，占固定的38字节。
   - Page Header ，表示数据页专有的一些信息，占固定的56个字节。
   - Infimum + Supremum ，两个虚拟的伪记录，分别表示页中的最小和最大记录，占固定的 26 个字节。
   - User Records ：真实存储我们插入的记录的部分，大小不固定。
   - Free Space ：页中尚未使用的部分，大小不确定。

- `Page Directory`：页中的某些记录相对位置，也就是各个槽在页面中的地址偏移量，大小不固定，插入的记录越多，这个部分占用的空间越多。
- `File Trailer`：用于检验页是否完整的部分，占用固定的8个字节。

3. 每个记录的头信息中都有一个 `next_record` 属性，从而使页中的所有记录串联成一个 `单链表`。
4. `InnoDB` 会为把页中的记录划分为若干个组，每个组的最后一个记录的地址偏移量作为一个 `槽`，存放在 `Page Directory` 中，所以在一个页中根据主键查找记录是非常快的，分为两步：

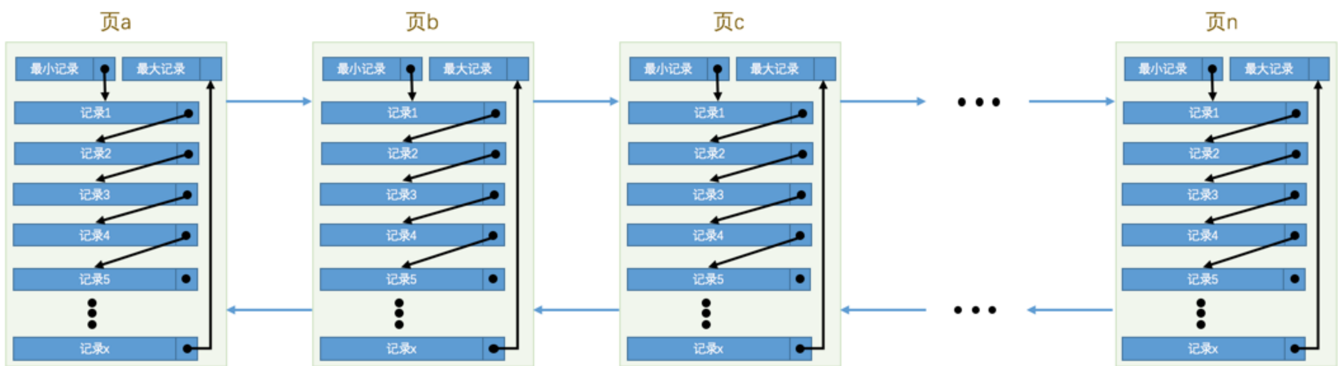
    - 通过二分法确定该记录所在的槽。
    - 通过记录的next_record属性遍历该槽所在的组中的各个记录。

5. 每个数据页的 `File Header` 部分都有上一个和下一个页的编号，所以所有的数据页会组成一个 `双链表`。
6. 为保证从内存中同步到磁盘的页的完整性，在页的首部和尾部都会存储页中数据的校验和和页面最后修改时对应的 `LSN` 值，如果首部和尾部的校验和和 `LSN` 值校验不成功的话，就说明同步过程出现了问题。

6 第6章 快速查询的秘籍-B+树索引

标签： MySQL是怎样运行的

前边我们详细唠叨了 `InnoDB` 数据页的7个组成部分，知道了各个数据页可以组成一个 `双向链表`，而每个数据页中的记录会按照主键值从小到大的顺序组成一个 `单向链表`，每个数据页都会为存储在它里边儿的记录生成一个 `页目录`，在通过主键查找某条记录的时候可以在 `页目录` 中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录（如果你对这段话有一丁点儿疑惑，那么接下来的部分不适合你，返回去看一下数据页结构吧）。页和记录的关系示意图如下：



其中页a、页b、页c ... 页n 这些页可以不在物理结构上相连，只要通过双向链表相关联即可。

6.1 没有索引的查找

本集的主题是 `索引`，在正式介绍 `索引` 之前，我们需要了解一下没有索引的时候是怎么查找记录的。为了方便大家理解，我们下边先只唠叨搜索条件为对某个列精确匹配的情况，所谓精确匹配，就是搜索条件中用等于 `=` 连接起的表达式，比如这样：

```
SELECT [列名列表] FROM 表名 WHERE 列名 = xxx;
```

6.1.1 在一个页中的查找

假设目前表中的记录比较少，所有的记录都可以被存放到一个页中，在查找记录的时候可以根据搜索条件的不同分为两种情况：

- 以主键为搜索条件

这个查找过程我们已经很熟悉了，可以在 页目录 中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录。

- 以其他列作为搜索条件

  对非主键列的查找的过程可就不这么幸运了，因为在数据页中并没有对非主键列建立所谓的 页目录 ，所以我们无法通过二分法快速定位相应的 槽 。这种情况下只能从 最小记录 开始依次遍历单链表中的每条记录，然后对比每条记录是不是符合搜索条件。很显然，这种查找的效率是非常低的。

6.1.2 在很多页中查找

大部分情况下我们表中存放的记录都是非常多的，需要好多的数据页来存储这些记录。在很多页中查找记录的话可以分为两个步骤：

1. 定位到记录所在的页。
2. 从所在的页内中查找相应的记录。

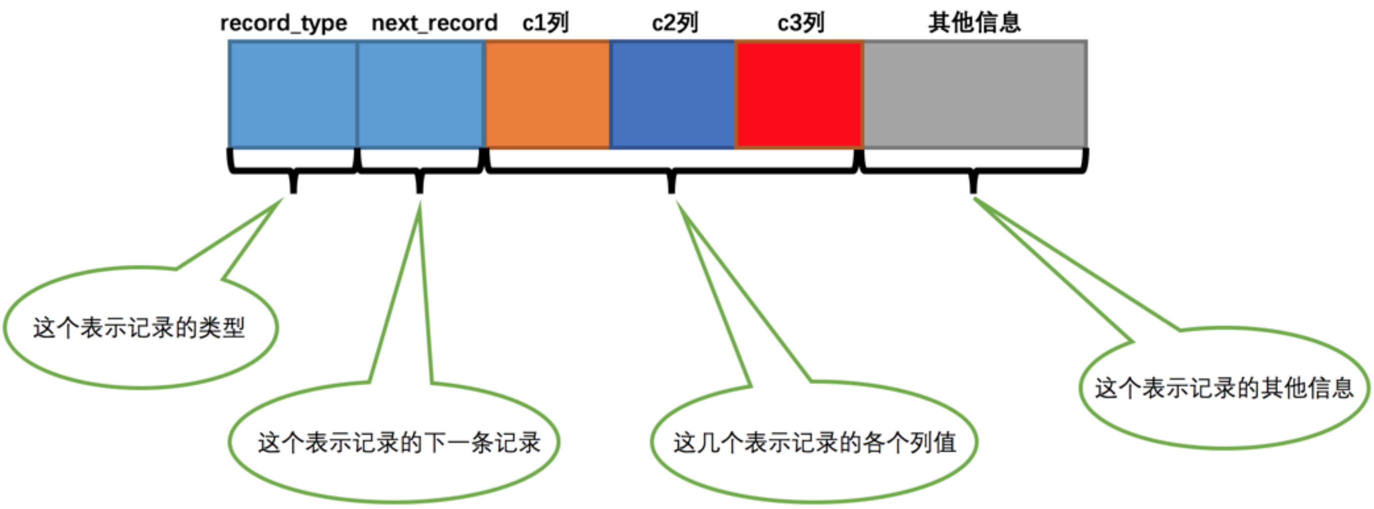
在没有索引的情况下，不论是根据主键列或者其他列的值进行查找，由于我们并不能快速的定位到记录所在的页，所以只能从第一个页沿着双向链表一直往下找，在每一个页中根据我们刚刚唠叨过的查找方式去查找指定的记录。因为要遍历所有的数据页，所以这种方式显然是超级耗时的，如果一个表有一亿条记录，使用这种方式去查找记录那要等到猴年马月才能等到查找结果。所以祖国和人民都在期盼一种能高效完成搜索的方法， 索引 同志就要亮相登台了。

6.2 索引

为了故事的顺利发展，我们先建一个表：

```
mysql> CREATE TABLE index_demo(
    ->     c1 INT,
    ->     c2 INT,
    ->     c3 CHAR(1),
    ->     PRIMARY KEY(c1)
    -> ) ROW_FORMAT = Compact;
Query OK, 0 rows affected (0.03 sec)
```

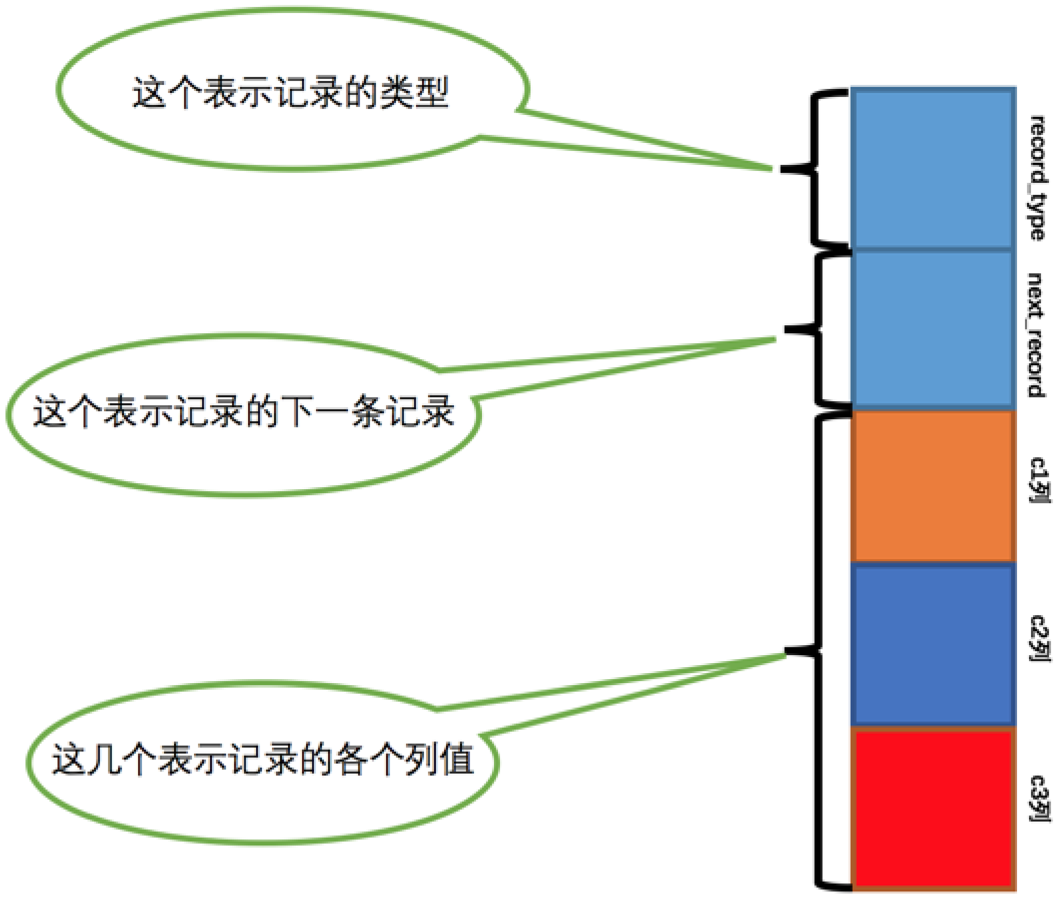
这个新建的 index_demo 表中有2个 INT 类型的列，1个 CHAR(1) 类型的列，而且我们规定了 c1 列为主键，这个表使用 Compact 行格式来实际存储记录的。为了我们理解上的方便，我们简化了一下 index_demo 表的行格式示意图：



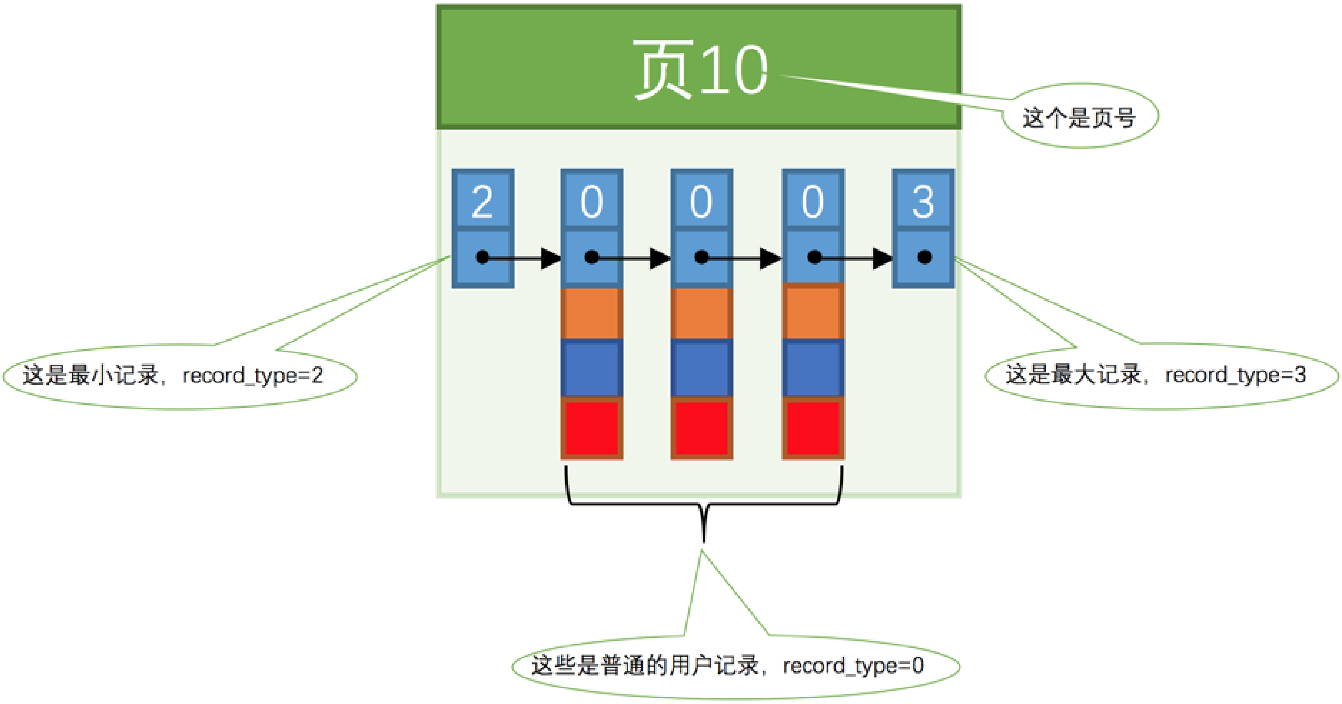
我们只在示意图里展示记录的这几个部分：

- record_type ：记录头信息的一项属性，表示记录的类型， 0 表示普通记录、 2 表示最小记录、 3 表示最大记录、 1 我们还没用过，等会再说～
- next_record ：记录头信息的一项属性，表示下一条地址相对于本条记录的地址偏移量，为了方便大家理解，我们都会用箭头来表明下一条记录是谁。
- 各个列的值 ：这里只记录在 index_demo 表中的三个列，分别是 c1 、 c2 和 c3 。
- 其他信息 ：除了上述3种信息以外的所有信息，包括其他隐藏列的值以及记录的额外信息。

为了节省篇幅，我们之后的示意图中会把记录的 其他信息 这个部分省略掉，因为它占地方并且不会有什么观赏效果。另外，为了方便理解，我们觉得把记录竖着放看起来感觉更好，所以将记录格式示意图的 其他信息 去掉并把它竖起来的效果就是这样：



把一些记录放到页里边的示意图就是：



6.2.1 一个简单的索引方案

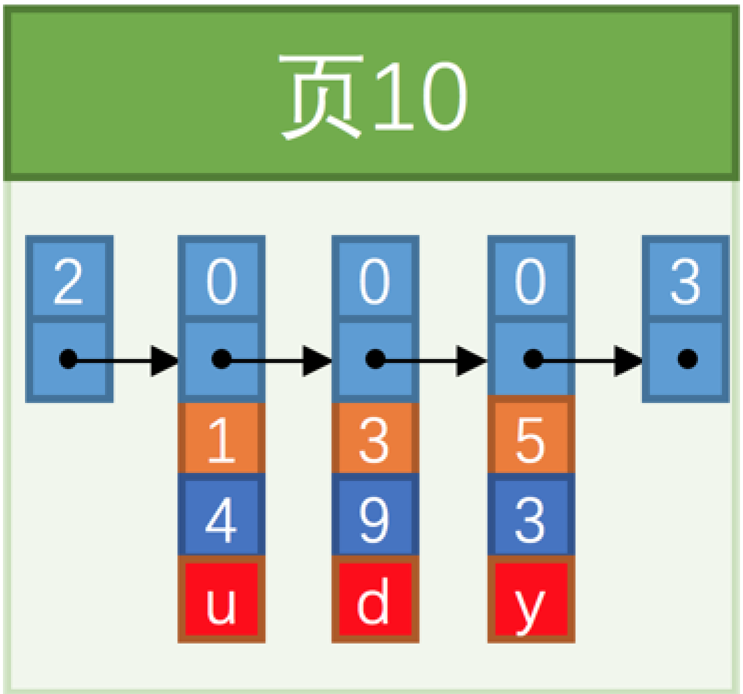
回到正题，我们在根据某个搜索条件查找一些记录时为什么要遍历所有的数据页呢？因为各个页中的记录并没有规律，我们并不知道我们的搜索条件匹配哪些页中的记录，所以 ***不得不*** 依次遍历所有的数据页。所以如果我们想快速的定位到需要查找的记录在哪些数据页中该咋办？还记得我们为根据主键值快速定位一条记录在页中的位置而设立的页目录么？我们也可以想办法为快速定位记录所在的数据页而建立一个别的目录，建这个目录必须完成下边这些事儿：

- 下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。

  为了故事的顺利发展，我们这里需要做一个假设：假设我们的每个数据页最多能存放3条记录（实际上一个数据页非常大，可以存放下好多记录）。有了这个假设之后我们向 `index_demo` 表插入3条记录：

  ```
  mysql> INSERT INTO index_demo VALUES(1, 4, 'u'), (3, 9, 'd'), (5, 3, 'y');
  Query OK, 3 rows affected (0.01 sec)
  Records: 3  Duplicates: 0  Warnings: 0
  ```

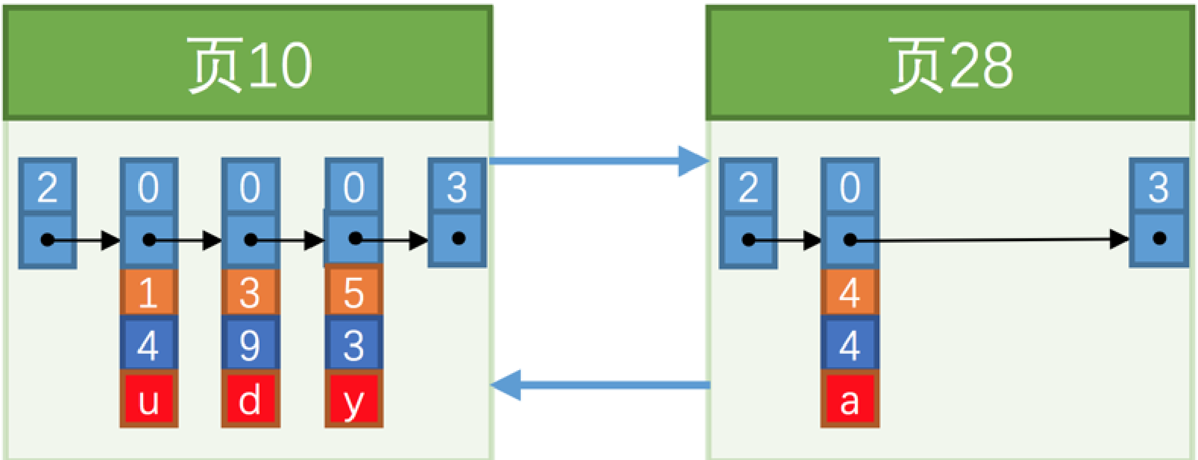
  那么这些记录已经按照主键值的大小串联成一个单向链表了，如图所示：



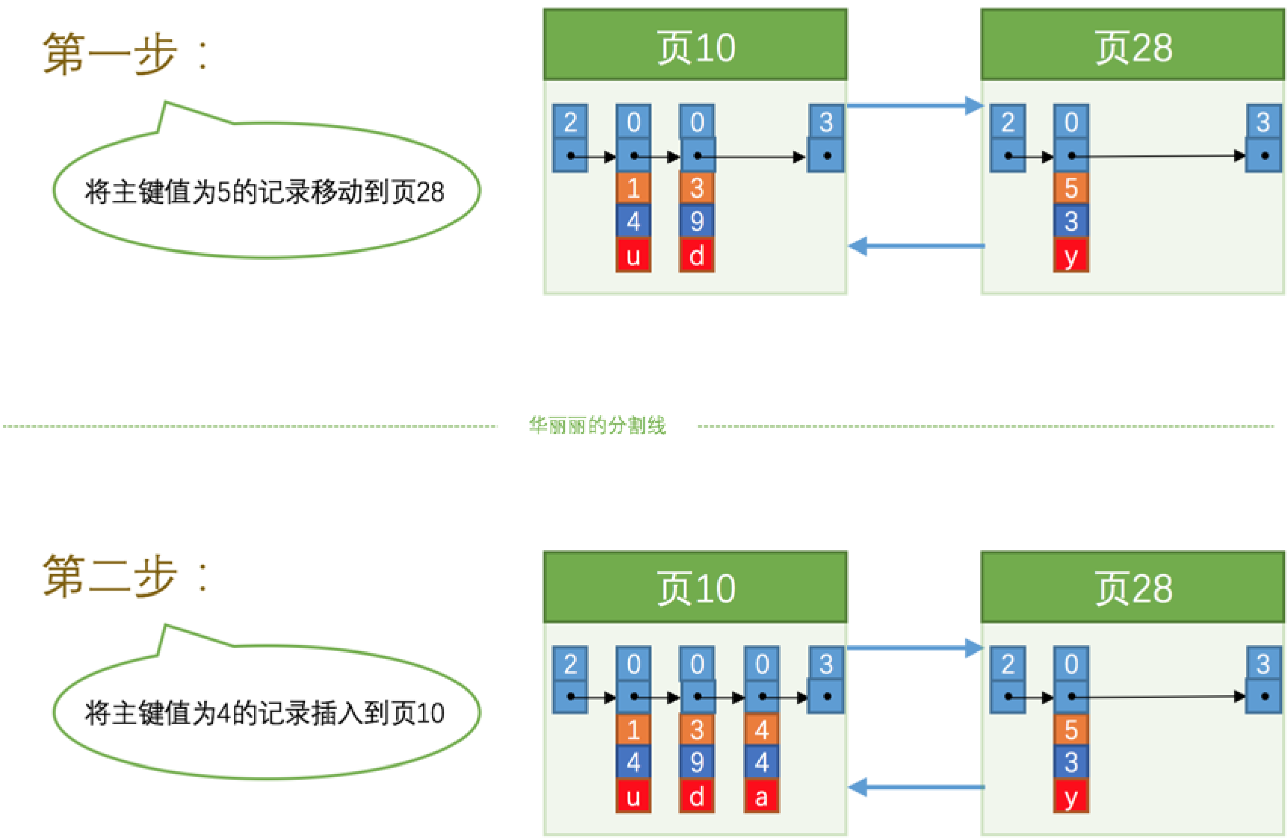
从图中可以看出来， index_demo 表中的3条记录都被插入到了编号为 10 的数据页中了。此时我们再来插入一条记录：

```
mysql> INSERT INTO index_demo VALUES(4, 4, 'a');
Query OK, 1 row affected (0.00 sec)
```

因为 页10 最多只能放3条记录，所以我们不得不再分配一个新页：



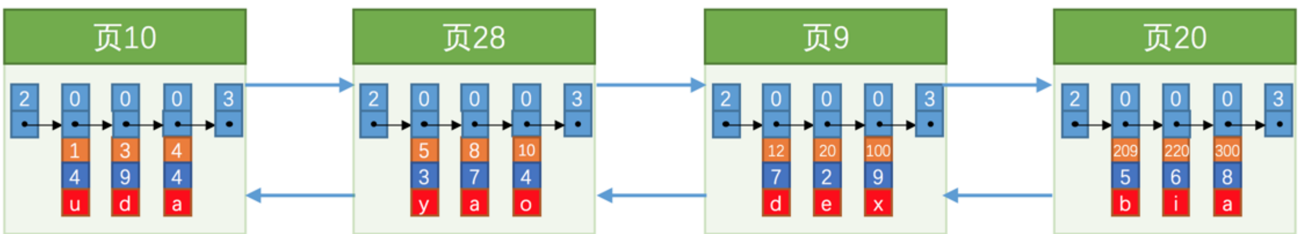
咦？怎么分配的页号是 28 呀，不应该是 11 么？再次强调一遍，新分配的数据页编号可能并不是连续的，也就是说我们使用的这些页在存储空间里可能并不挨着。它们只是通过维护着上一个页和下一个页的编号而建立了链表关系。另外， 页10 中用户记录最大的主键值是 5 ，而 页28 中有一条记录的主键值是 4 ，因为 5 > 4 ，所以这就不符合下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值的要求，所以在插入主键值为 4 的记录的时候需要伴随着一次记录移动，也就是把主键值为 5 的记录移动到 页28 中，然后再把主键值为 4 的记录插入到 页10 中，这个过程的示意图如下：



这个过程表明了在对页中的记录进行增删改操作的过程中，我们必须通过一些诸如记录移动的操作来始终保证这个状态一直成立：下一个数据页中用户记录的主键值必须大于上一个页中用户记录的主键值。这个过程我们也可以称为 页分裂 。

- 给所有的页建立一个目录项。

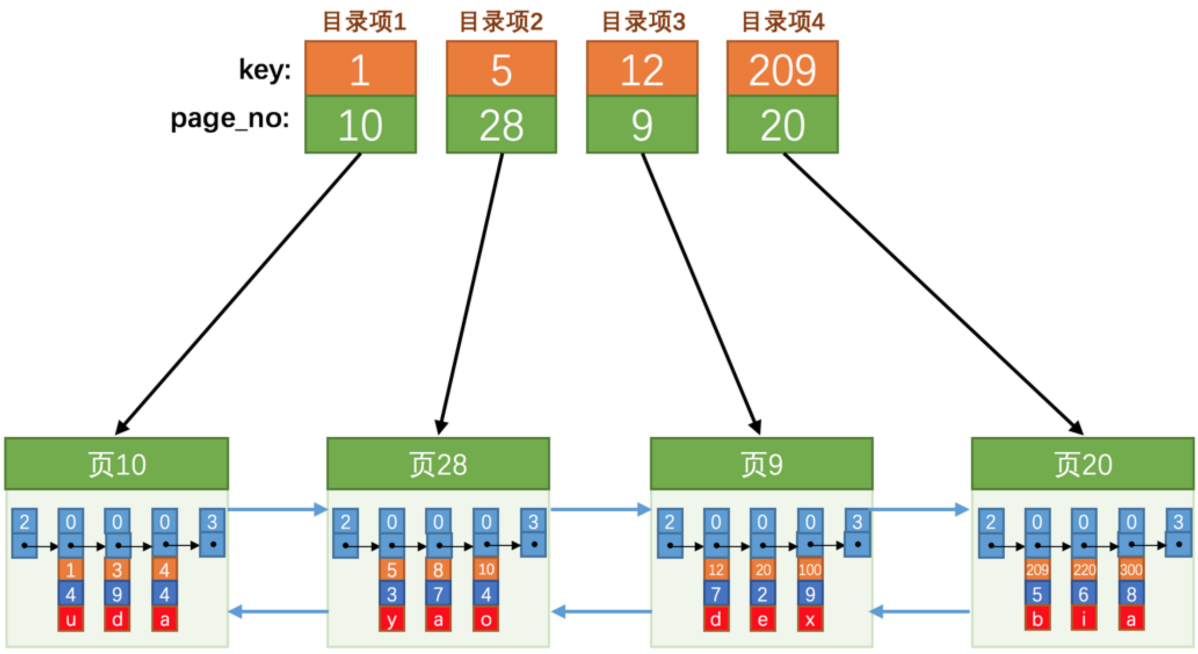
由于数据页的编号可能并不是连续的，所以在向 index_demo 表中插入许多条记录后，可能是这样的效果：



因为这些 16KB 的页在物理存储上可能并不挨着，所以如果想从这么多页中根据主键值快速定位某些记录所在的页，我们需要给它们做个目录，每个页对应一个目录项，每个目录项包括下边两个部分：

- 页的用户记录中最小的主键值，我们用 key 来表示。
- 页号，我们用 page_no 表示。

所以我们为上边几个页做好的目录就像这样子：



以 页28 为例，它对应 目录项2 ，这个目录项中包含着该页的页号 28 以及该页中用户记录的最小主键值 5 。我们只需要把几个目录项在物理存储器上连续存储，比如把他们放到一个数组里，就可以实现根据主键值快速查找某条记录的功能了。比方说我们想找主键值为 20 的记录，具体查找过程分两步：

- 先从目录项中根据二分法快速确定出主键值为 20 的记录在 目录项3 中（因为 12 < 20 < 209 ），它对应的页是 页9 。
- 再根据前边说的在页中查找记录的方式去 页9 中定位具体的记录。

至此，针对数据页做的简易目录就搞定了。不过忘了说了，这个 目录 有一个别名，称为 索引 。

6.2.2 InnoDB中的索引方案

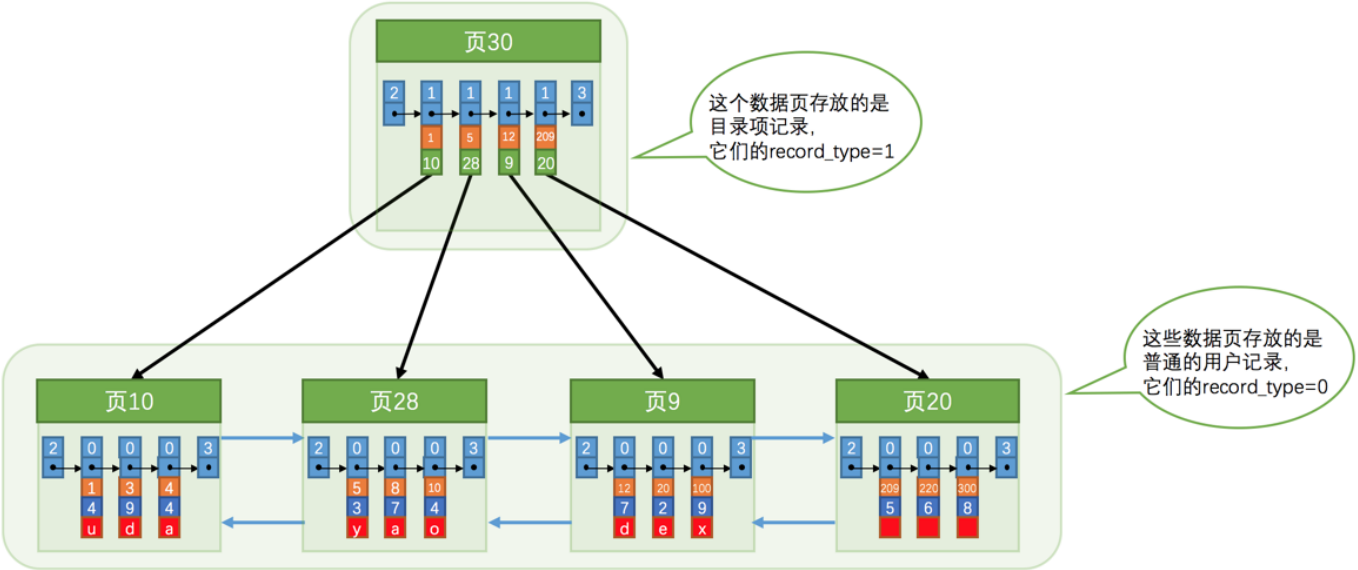
上边之所以称为一个简易的索引方案，是因为我们为了在根据主键值进行查找时使用二分法快速定位具体的目录项而假设所有目录项都可以在物理存储器上连续存储，但是这样做有几个问题：

- InnoDB 是使用页来作为管理存储空间的基本单位，也就是最多能保证 16KB 的连续存储空间，而随着表中记录数量的增多，需要非常大的连续的存储空间才能把所有的目录项都放下，这对记录数量非常多的表是不现实的。
- 我们时常会对记录进行增删，假设我们把 页28 中的记录都删除了， 页28 也就没有存在的必要了，那意味着 目录项2 也就没有存在的必要了，这就需要把 目录项2 后的目录项都向前移动一下，这种牵一发而动全身的设计不是什么好主意～

所以，设计 InnoDB 的大叔们需要一种可以灵活管理所有 目录项 的方式。他们灵光乍现，忽然发现这些 目录项 其实长得跟我们的用户记录差不多，只不过 目录项 中的两个列是 主键 和 页号 而已，所以他们复用了之前存储用户记录的数据页来存储目录项，为了和用户记录做一下区分，我们把这些用来表示目录项的记录称为 目录项记录 。那 InnoDB 怎么区分一条记录是普通的 用户记录 还是 目录项记录 呢？别忘了记录头信息里的 record_type 属性，它的各个取值代表的意思如下：

- 0 ：普通的用户记录
- 1 ：目录项记录
- 2 ：最小记录
- 3 ：最大记录

哈哈，原来这个值为 1 的 record_type 是这个意思呀，我们把前边使用到的目录项放到数据页中的样子就是这样：



从图中可以看出来，我们新分配了一个编号为 30 的页来专门存储 目录项记录 。这里再次强调一遍 目录项记录 和普通的 用户记录 的不同点：

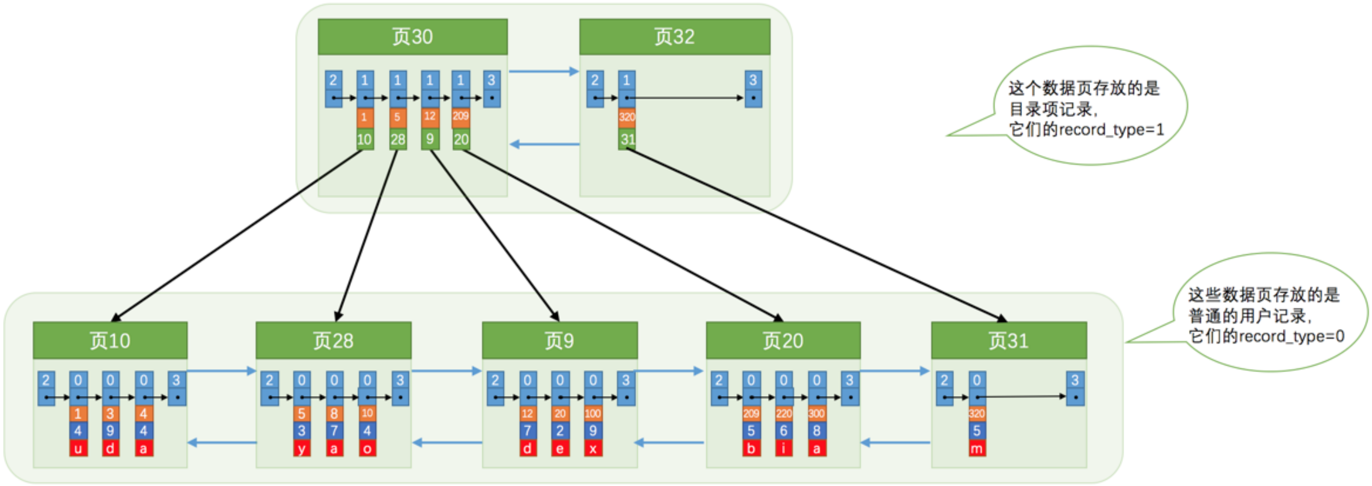
- 目录项记录 的 record_type 值是1，而普通用户记录的 record_type 值是0。
- 目录项记录 只有主键值和页的编号两个列，而普通的用户记录的列是用户自己定义的，可能包含很多列，另外还有 InnoDB 自己添加的隐藏列。
- 还记得我们之前在唠叨记录头信息的时候说过一个叫 min_rec_mask 的属性么，只有在存储 目录项记录 的页中的主键值最小的 目录项记录 的 min_rec_mask 值为 1 ，其他别的记录的 min_rec_mask 值都是 0 。

除了上述几点外，这两者就没啥差别了，它们用的是一样的数据页（页面类型都是 0x45BF ，这个属性在 File Header 中，忘了的话可以翻到前边的文章看），页的组成结构也是一样一样的（就是我们前边介绍过的7个部分），都会为主键值生成 Page Directory （页目录），从而在按照主键值进行查找时可以使用二分法来加快查询速度。现在以查找主键为 20 的记录为例，根据某个主键值去查找记录的步骤就可以大致拆分成下边两步：

1. 先到存储 目录项记录 的页，也就是页 30 中通过二分法快速定位到对应目录项，因为 12 < 20 < 209 ，所以定位到对应的记录所在的页就是 页9 。
2. 再到存储用户记录的 页9 中根据二分法快速定位到主键值为 20 的用户记录。

虽然说 目录项记录 中只存储主键值和对应的页号，比用户记录需要的存储空间小多了，但是不论怎么说一个页只有 16KB 大小，能存放的 目录项记录 也是有限的，那如果表中的数据太多，以至于一个数据页不足以存放所有的 目录项记录 ，该咋办呢？

当然是再多整一个存储 目录项记录 的页喽～ 为了大家更好的理解新分配一个 目录项记录 页的过程，我们假设一个存储 目录项记录 的页最多只能存放4条 目录项记录 （请注意是假设哦，真实情况下可以存放好多条的），所以如果此时我们再向上图中插入一条主键值为 320 的用户记录的话，那就需要分配一个新的存储 目录项记录 的页喽：



从图中可以看出，我们插入了一条主键值为 320 的用户记录之后需要两个新的数据页：

- 为存储该用户记录而新生成了 页31 。
- 因为原先存储 目录项记录 的 页30 的容量已满（我们前边假设只能存储4条 目录项记录 ），所以不得不需要一个新的 页32 来存放 页31 对应的目录项。

现在因为存储 目录项记录 的页不止一个，所以如果我们想根据主键值查找一条用户记录大致需要3个步骤，以查找主键值为 20 的记录为例：

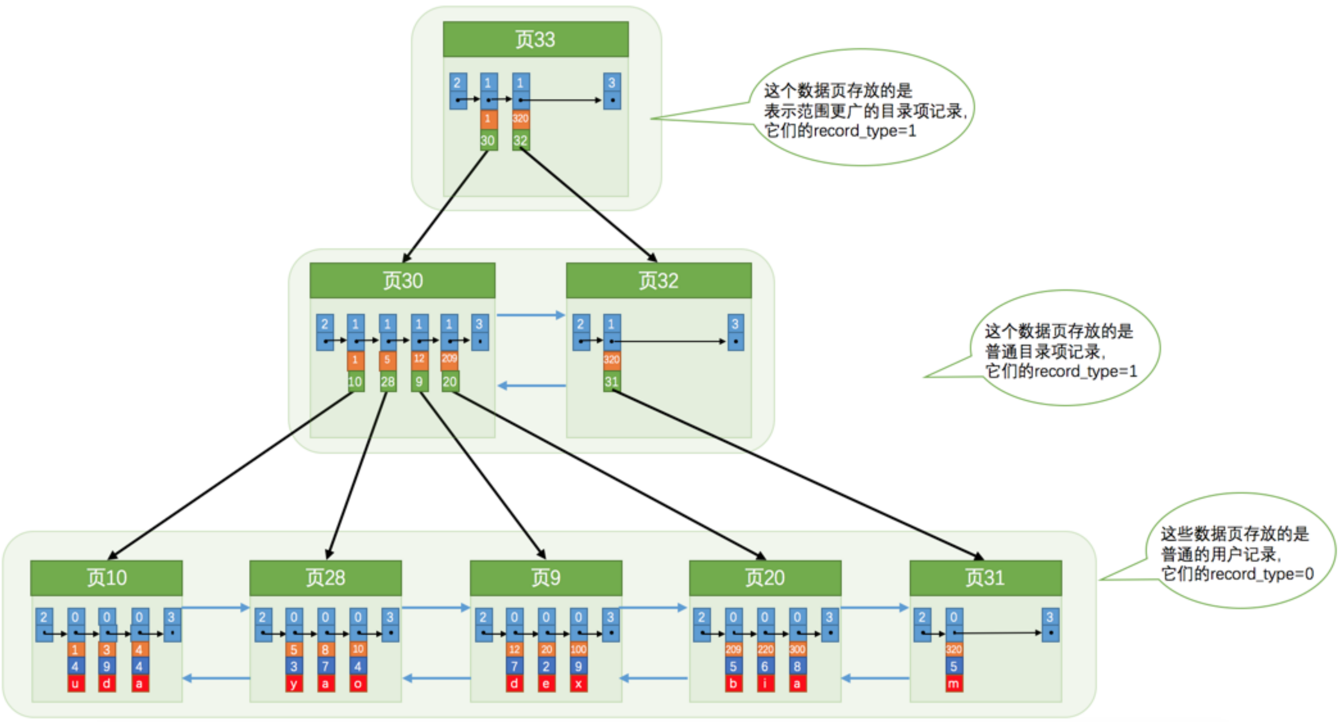
1. 确定 目录项记录 页

   我们现在的存储 目录项记录 的页有两个，即 页30 和 页32 ，又因为 页30 表示的目录项的主键值的范围是 [1, 320) ， 页32 表示的目录项的主键值不小于 320 ，所以主键值为 20 的记录对应的目录项记录在 页30 中。
2. 通过 目录项记录 页确定用户记录真实所在的页。

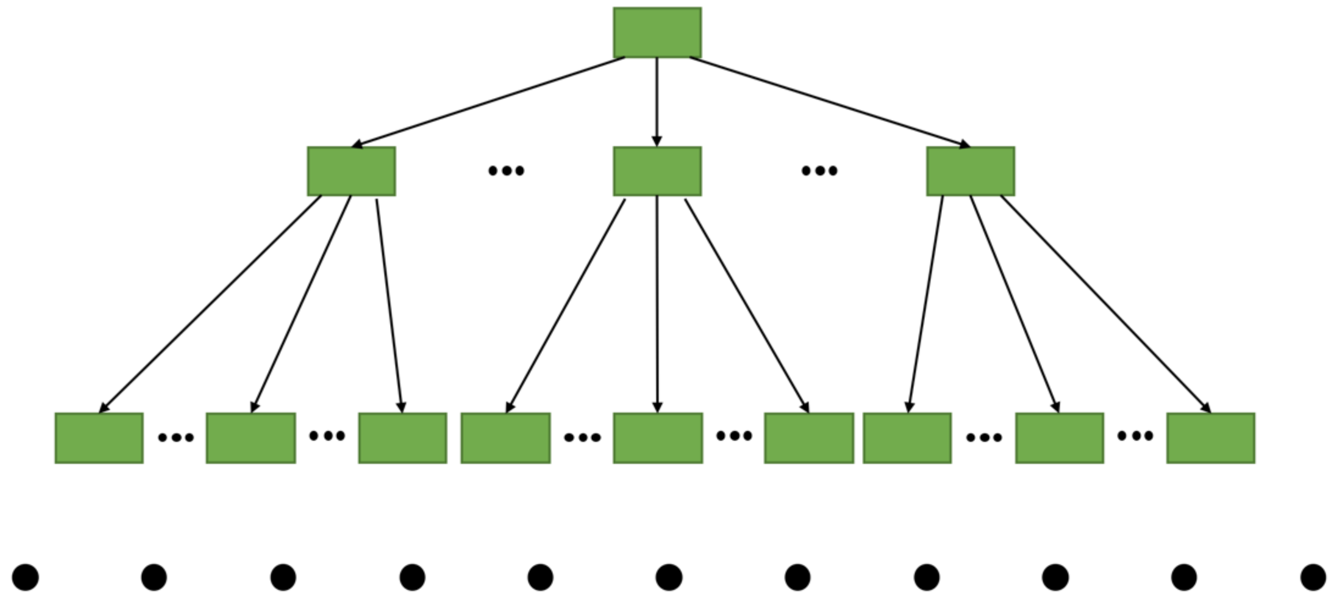
   在一个存储 目录项记录 的页中通过主键值定位一条目录项记录的方式说过了，不赘述了～
3. 在真实存储用户记录的页中定位到具体的记录。

   在一个存储用户记录的页中通过主键值定位一条用户记录的方式已经说过200遍了，你再不会我就，我就，我就求你到上一篇唠叨数据页结构的文章中多看几遍，求你了～

那么问题来了，在这个查询步骤的第1步中我们需要定位存储 目录项记录 的页，但是这些页在存储空间中也可能不挨着，如果我们表中的数据非常多则会产生很多存储 目录项记录 的页，那我们怎么根据主键值快速定位一个存储 目录项记录 的页呢？其实也简单，为这些存储 目录项记录 的页再生成一个更高级的目录，就像是一个多级目录一样，大目录里嵌套小目录，小目录里才是实际的数据，所以现在各个页的示意图就是这样子：



如图，我们生成了一个存储更高级目录项的 页33 ，这个页中的两条记录分别代表 页30 和 页32 ，如果用户记录的主键值在 [1, 320) 之间，则到 页30 中查找更详细的 目录项记录 ，如果主键值不小于 320 的话，就到 页32 中查找更详细的 目录项记录 。不过这张图好漂亮喔，随着表中记录的增加，这个目录的层级会继续增加，如果简化一下，那么我们可以用下边这个图来描述它：



这玩意儿像不像一个倒过来的 树 呀，上头是树根，下头是树叶！其实这是一种组织数据的形式，或者说是一种数据结构，它的名称是 B+ 树。

> 小贴士：
> 为啥叫`B+`呢，`B`树是个啥？喔对不起，这不是我们讨论的范围，你可以去找一本数据结构或算法的书来看。什么？数据结构的书看不懂？等我～

不论是存放用户记录的数据页，还是存放目录项记录的数据页，我们都把它们存放到 B+ 树这个数据结构中了，所以我们也称这些数据页为 节点 。从图中可以看出来，我们的实际用户记录其实都存放在B+树的最底层的节点上，这些节点也被称为 叶子节点 或 叶节点 ，其余用来存放 目录项 的节点称为 非叶子节点 或者 内节点 ，其中 B+ 树最上边的那个节点也称为 根节点 。

从图中可以看出来，一个 B+ 树的节点其实可以分成好多层，设计 InnoDB 的大叔们为了讨论方便，规定最下边的那层，也就是存放我们用户记录的那层为第 0 层，之后依次往上加。之前的讨论我们做了一个非常极端的假设：存放用户记录的页最多存放3条记录，存放目录项记录的页最多存放4条记录。其实真实环境中一个页存放的记录数量是非常大的，假设，假设，假设所有存放用户记录的叶子节点代表的数据页可以存放100条用户记录，所有存放目录项记录的内节点代表的数据页可以存放1000条目录项记录，那么：

- 如果 B+ 树只有1层，也就是只有1个用于存放用户记录的节点，最多能存放 100 条记录。
- 如果 B+ 树有2层，最多能存放 1000×100=100000 条记录。
- 如果 B+ 树有3层，最多能存放 1000×1000×100=100000000 条记录。
- 如果 B+ 树有4层，最多能存放 1000×1000×1000×100=100000000000 条记录。哇咔咔~这么多的记录！！！

你的表里能存放 100000000000 条记录么？所以一般情况下，我们用到的 B+ 树都不会超过4层，那我们通过主键值去查找某条记录最多只需要做4个页面内的查找（查找3个目录项页和一个用户记录页），又因为在每个页面内有所谓的 Page Directory （页目录），所以在页面内也可以通过二分法实现快速定位记录，这不是很牛么，哈哈！

6.2.2.1 聚簇索引

我们上边介绍的 B+ 树本身就是一个目录，或者说本身就是一个索引。它有两个特点：

1. 使用记录主键值的大小进行记录和页的排序，这包括三个方面的含义：

    - 页内的记录是按照主键的大小顺序排成一个单向链表。
    - 各个存放用户记录的页也是根据页中用户记录的主键大小顺序排成一个双向链表。
    - 存放目录项记录的页分为不同的层次，在同一层次中的页也是根据页中目录项记录的主键大小顺序排成一个双向链表。
2. B+ 树的叶子节点存储的是完整的用户记录。

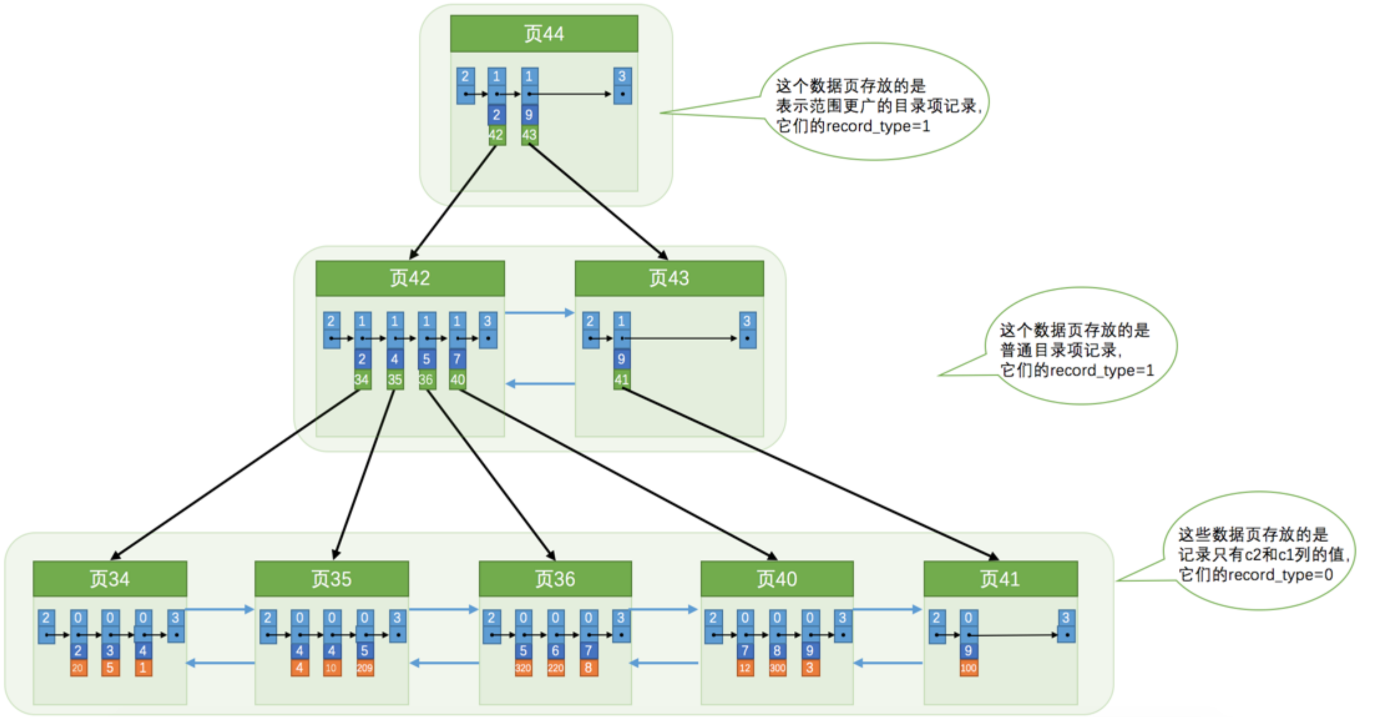
    所谓完整的用户记录，就是指这个记录中存储了所有列的值（包括隐藏列）。

我们把具有这两种特性的 B+ 树称为 聚簇索引 ，所有完整的用户记录都存放在这个 聚簇索引 的叶子节点处。这种 聚簇索引 并不需要我们在 MySQL 语句中显式的使用 INDEX 语句去创建（后边会介绍索引相关的语句）， InnoDB 存储引擎会自动的为我们创建聚簇索引。另外有趣的一点是，在 InnoDB 存储引擎中， 聚簇索引 就是数据的存储方式（所有的用户记录都存储在了 叶子节点 ），也就是所谓的索引即数据，数据即索引。

6.2.2.2 二级索引

大家有木有发现，上边介绍的 聚簇索引 只能在搜索条件是主键值时才能发挥作用，因为 B+ 树中的数据都是按照主键进行排序的。那如果我们想以别的列作为搜索条件该咋办呢？难道只能从头到尾沿着链表依次遍历记录么？

不，我们可以多建几棵 B+ 树，不同的 B+ 树中的数据采用不同的排序规则。比方说我们用 c2 列的大小作为数据页、页中记录的排序规则，再建一棵 B+ 树，效果如下图所示：



这个 B+ 树与上边介绍的聚簇索引有几处不同:

- 使用记录 c2 列的大小进行记录和页的排序，这包括三个方面的含义:
  - 页内的记录是按照 c2 列的大小顺序排成一个单向链表。
  - 各个存放用户记录的页也是根据页中记录的 c2 列大小顺序排成一个双向链表。
  - 存放目录项记录的页分为不同的层次，在同一层次中的页也是根据页中目录项记录的 c2 列大小顺序排成一个双向链表。
- B+ 树的叶子节点存储的并不是完整的用户记录，而只是 c2列+主键 这两个列的值。
- 目录项记录中不再是 主键+页号 的搭配，而变成了 c2列+页号 的搭配。

所以如果我们现在想通过 c2 列的值查找某些记录的话就可以使用我们刚刚建好的这个 B+ 树了。以查找 c2 列的值为 4 的记录为例，查找过程如下:

1. 确定 目录项记录 页

   根据 根页面 ，也就是 页44 ，可以快速定位到 目录项记录 所在的页为 页42 （因为 2 < 4 < 9 ）。
2. 通过 目录项记录 页确定用户记录真实所在的页。

   在 页42 中可以快速定位到实际存储用户记录的页，但是由于 c2 列并没有唯一性约束，所以 c2 列值为 4 的记录可能分布在多个数据页中，又因为 2 < 4 ≤ 4 ，所以确定实际存储用户记录的页在 页34 和 页35 中。
3. 在真实存储用户记录的页中定位到具体的记录。

   到 页34 和 页35 中定位到具体的记录。
4. 但是这个 B+ 树的叶子节点中的记录只存储了 c2 和 c1 （也就是 主键 ）两个列，所以我们必须再根据主键值去聚簇索引中再查找一遍完整的用户记录。

各位各位，看到步骤4的操作了么？我们根据这个以 c2 列大小排序的 B+ 树只能确定我们要查找记录的主键值，所以如果我们想根据 c2 列的值查找到完整的用户记录的话，仍然需要到 聚簇索引 中再查一遍，这个过程也被称为 回表 。也就是根据 c2 列的值查询一条完整的用户记录需要使用到 2 棵 B+ 树！！！

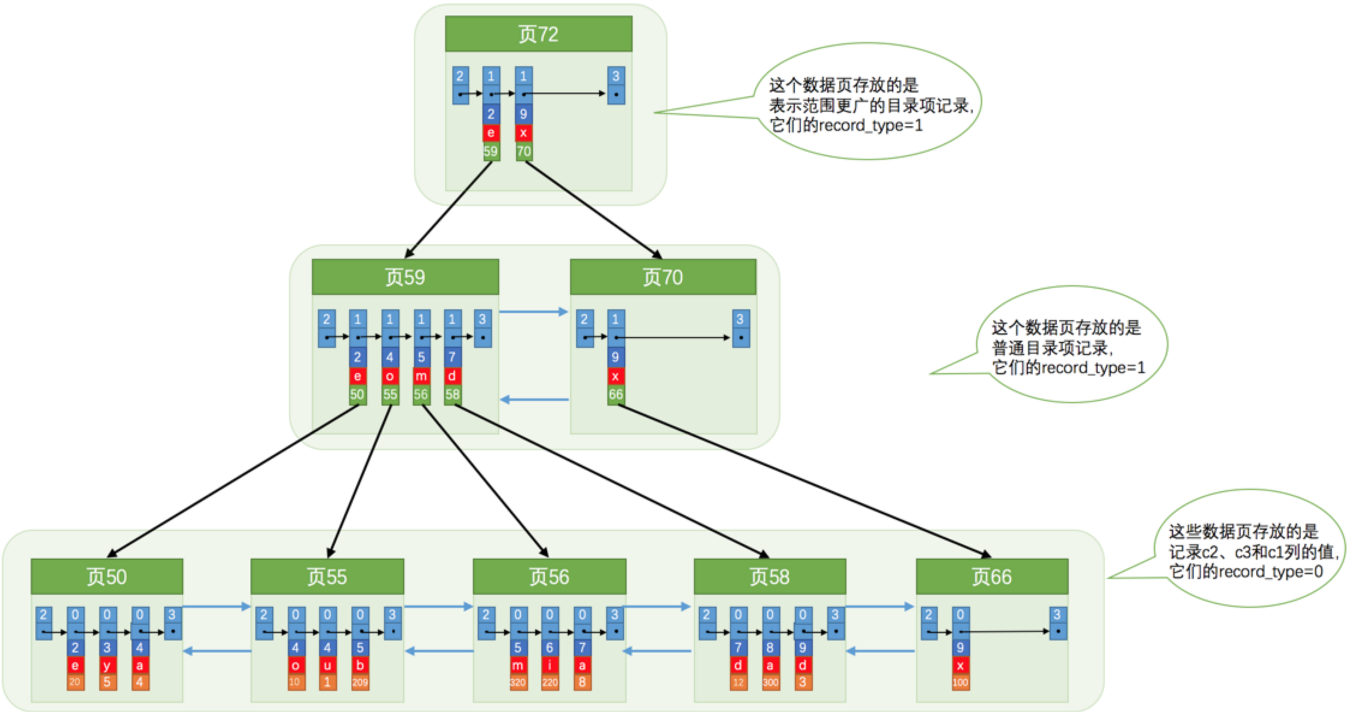
为什么我们还需要一次 回表 操作呢？直接把完整的用户记录放到 叶子节点 不就好了么？你说的对，如果把完整的用户记录放到 叶子节点 是可以不用 回表 ，但是太占地方了呀~相当于每建立一棵 B+ 树都需要把所有的用户记录再都拷贝一遍，这就有点太浪费存储空间了。因为这种按照 非主键列 建立的 B+ 树需要一次 回表 操作才可以定位到完整的用户记录，所以这种 B+ 树也被称为 二级索引 （英文名 secondary index ），或者 辅助索引 。由于我们使用的是 c2 列的大小作为 B+ 树的排序规则，所以我们也称这个 B+ 树为为c2列建立的索引。

联合索引

我们也可以同时以多个列的大小作为排序规则，也就是同时为多个列建立索引，比方说我们想让 B+ 树按照 c2 和 c3 列的大小进行排序，这个包含两层含义：

- 先把各个记录和页按照 c2 列进行排序。
- 在记录的 c2 列相同的情况下，采用 c3 列进行排序

为 c2 和 c3 列建立的索引的示意图如下：



如图所示，我们需要注意一下几点：

- 每条 目录项记录 都由 c2 、 c3 、 页号 这三个部分组成，各条记录先按照 c2 列的值进行排序，如果记录的 c2 列相同，则按照 c3 列的值进行排序。
- B+ 树叶子节点处的用户记录由 c2 、 c3 和主键 c1 列组成。

千万要注意一点，以c2和c3列的大小为排序规则建立的B+树称为联合索引，本质上也是一个二级索引。它的意思与分别为c2和c3列分别建立索引的表述是不同的，不同点如下：

- 建立 联合索引 只会建立如上图一样的1棵 B+ 树。
- 为c2和c3列分别建立索引会分别以 c2 和 c3 列的大小为排序规则建立2棵 B+ 树。

6.2.3 InnoDB的B+树索引的注意事项

6.2.3.1 根页面万年不动窝

我们前边介绍 B+ 树索引的时候，为了大家理解上的方便，先把存储用户记录的叶子节点都画出来，然后接着画存储目录项记录的内节点，实际上 B+ 树的形成过程是这样的：

- 每当为某个表创建一个 B+ 树索引（聚簇索引不是人为创建的，默认就有）的时候，都会为这个索引创建一个 根节点 页面。最开始表中没有数据的时候，每个 B+ 树索引对应的 根节点 中既没有用户记录，也没有目录项记录。
- 随后向表中插入用户记录时，先把用户记录存储到这个 根节点 中。

- 当 根节点 中的可用空间用完时继续插入记录，此时会将 根节点 中的所有记录复制到一个新分配的页，比如 页a 中，然后对这个新页进行 页分裂 的操作，得到另一个新页，比如 页b 。这时新插入的记录根据键值（也就是聚簇索引中的主键值，二级索引中对应的索引列的值）的大小就会被分配到 页a 或者 页b 中，而 根节点 便升级为存储目录项记录的页。

这个过程需要大家特别注意的是：一个B+树索引的根节点自诞生之日起，便不会再移动。这样只要我们对某个表建立一个索引，那么它的 根节点 的页号便会被记录到某个地方，然后凡是 InnoDB 存储引擎需要用到这个索引的时候，都会从那个固定的地方取出 根节点 的页号，从而来访问这个索引。

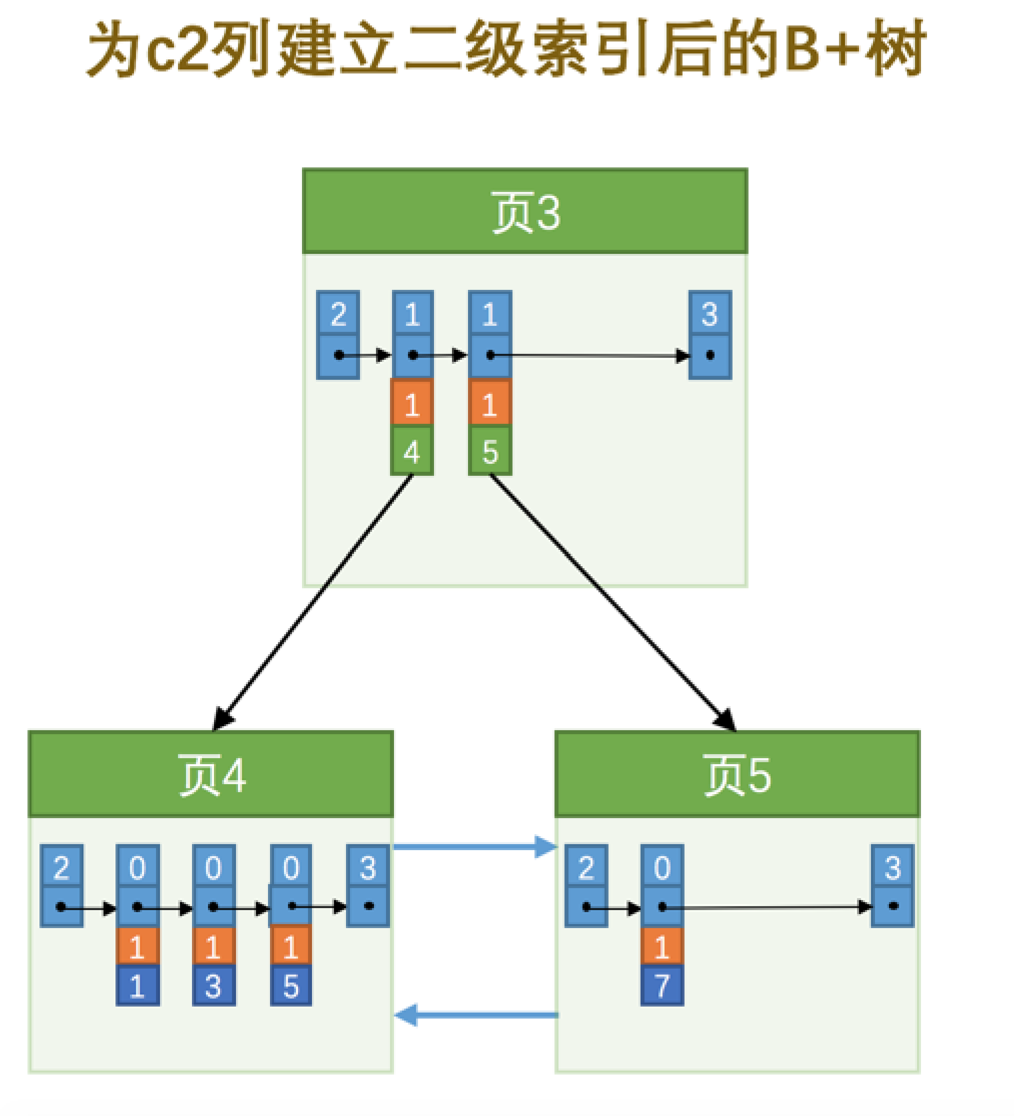
> 小贴士：
> 跟大家剧透一下，这个存储某个索引的根节点在哪个页面中的信息就是传说中的数据字典中的一项信息，关于更多数据字典的内容，后边会详细唠叨，别着急哈。

6.2.3.2 内节点中目录项记录的唯一性

我们知道 B+ 树索引的内节点中目录项记录的内容是 索引列 + 页号 的搭配，但是这个搭配对于二级索引来说有点儿不严谨。还拿 index_demo 表为例，假设这个表中的数据是这样的：

c1	c2	c3
1	1	'u'
3	1	'd'
5	1	'y'
7	1	'a'

如果二级索引中目录项记录的内容只是 索引列 + 页号 的搭配的话，那么为 c2 列建立索引后的 B+ 树应该长这样：

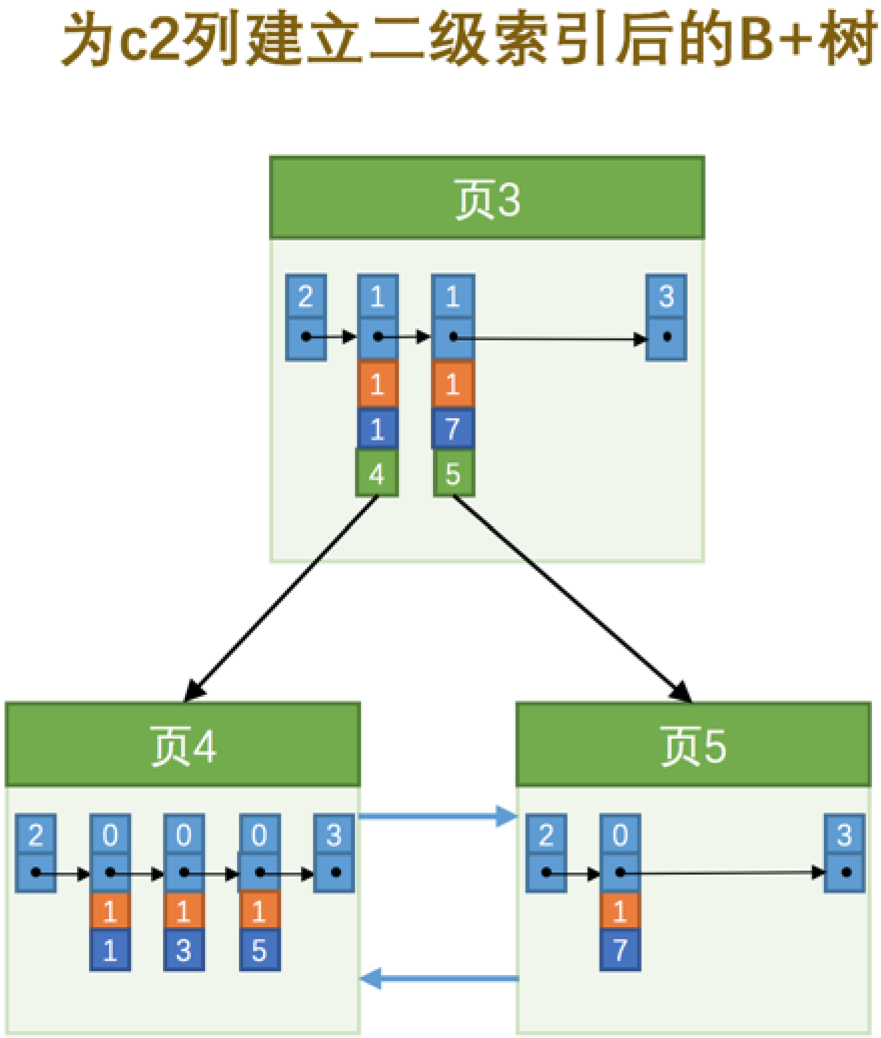


如果我们想新插入一行记录，其中 c1 、 c2 、 c3 的值分别是： 9 、 1 、 'c' ，那么在修改这个为 c2 列建立的二级索引对应的 B+ 树时便碰到了个大问题：由于 页3 中存储的目录项记录是由 c2列 + 页号 的值构成的， 页3 中的两条目录项记录对应的 c2 列的值都是 1 ，而我们新插入的这条记录的 c2 列的值也是 1 ，那我们这条新插入的记录到底应该放到 页4 中，还是应该放到 页5 中啊？答案是：对不起，懵逼了。

为了让新插入记录能找到自己在那个页里，我们需要保证在B+树的同一层内节点的目录项记录除 页号 这个字段以外是唯一的。所以对于二级索引的内节点的目录项记录的内容实际上是由三个部分构成的：

- 索引列的值
- 主键值
- 页号

也就是我们把 主键值 也添加到二级索引内节点中的目录项记录了，这样就能保证 B+ 树每一层节点中各条目录项记录除 页号 这个字段外是唯一的，所以我们为 c2 列建立二级索引后的示意图实际上应该是这样子的：



这样我们再插入记录（9， 1， ’c’）时，由于 页3 中存储的目录项记录是由 c2列 + 主键 + 页号 的值构成的，可以先把新记录的 c2 列的值和 页3 中各目录项记录的 c2 列的值作比较，如果 c2 列的值相同的话，可以接着比较主键值，因为 B+ 树同一层中不同目录项记录的 c2列 + 主键 的值肯定是不一样的，所以最后肯定能定位唯一的一条目录项记录，在本例中最后确定新记录应该被插入到 页5 中。

6.2.3.3 一个页面最少存储2条记录

我们前边说过一个B+树只需要很少的层级就可以轻松存储数亿条记录，查询速度杠杠的！这是因为B+树本质上就是一个大的多层级目录，每经过一个目录时都会过滤掉许多无效的子目录，直到最后访问到存储真实数据的目录。那如果一个大的目录中只存放一个子目录是个啥效果呢？那就是目录层级非常非常非常多，而且最后的那个存放真实数据的目录中只能存放一条记录。费了半天劲只能存放一条真实的用户记录？逗我呢？所以 InnoDB 的一个数据页至少可以存放两条记录，这也是我们之前唠叨记录行格式的时候说过一个结论（我们当时依据这个结论推导了表中只有一个列时该列在不发生行溢出的情况下最多能存储多少字节，忘了的话回去看看吧）。

6.2.4 MyISAM中的索引方案简单介绍

至此，我们介绍的都是 InnoDB 存储引擎中的索引方案，为了内容的完整性，以及各位可能在面试的时候遇到这类的问题，我们有必要再简单介绍一下 MyISAM 存储引擎中的索引方案。我们知道 InnoDB 中索引即数据，也就是聚簇索引的那棵 B+ 树的叶子节点中已经把所有完整的用户记录都包含了，而 MyISAM 的索引方案虽然也使用树形结构，但是却将索引和数据分开存储：

- 将表中的记录按照记录的插入顺序单独存储在一个文件中，称之为 数据文件 。这个文件并不划分为若干个数据页，有多少记录就往这个文件中塞多少记录就成了。我们可以通过行号而快速访问到一条记录。

MyISAM 记录也需要记录头信息来存储一些额外数据，我们以上边唠叨过的 index_demo 表为例，看一下这个表中的记录使用 MyISAM 作为存储引擎在存储空间中的表示：

0	记录头	1	4	u
1	记录头	3	9	d
2	记录头	5	3	y
3	记录头	4	4	a
4	记录头	100	9	x
5	记录头	8	7	a
6	记录头	209	5	b
7	记录头	300	8	a
8	记录头	20	2	e
9	记录头	10	4	o
10	记录头	12	7	d
11	记录头	220	6	i
12	记录头	320	5	m

由于在插入数据的时候并没有刻意按照主键大小排序，所以我们并不能在这些数据上使用二分法进行查找。

- 使用 MyISAM 存储引擎的表会把索引信息另外存储到一个称为 索引文件 的另一个文件中。 MyISAM 会单独为表的主键创建一个索引，只不过在索引的叶子节点中存储的不是完整的用户记录，而是 主键值 + 行号 的组合。也就是先通过索引找到对应的行号，再通过行号去找对应的记录！

  这一点和 InnoDB 是完全不相同的，在 InnoDB 存储引擎中，我们只需要根据主键值对 聚簇索引 进行一次查找就能找到对应的记录，而在 MyISAM 中却需要进行一次 回表 操作，意味着 MyISAM 中建立的索引相当于全部都是 二级索引 ！
- 如果有需要的话，我们也可以对其它的列分别建立索引或者建立联合索引，原理和 InnoDB 中的索引差不多，不过在叶子节点处存储的是 相应的列 + 行号 。这些索引也全部都是 二级索引 。

> 小贴士：
> MyISAM的行格式有定长记录格式（Static）、变长记录格式（Dynamic）、压缩记录格式（Compressed）。上边用到的index_demo表采用定长记录格式，也就是一条记录占用存储空间的大小是固定的，这样就可以轻松算出某条记录在数据文件中的地址偏移量。但是变长记录格式就不行了，MyISAM会直接在索引叶子节点处存储该条记录在数据文件中的地址偏移量。通过这个可以看出，MyISAM的回表操作是十分快速的，因为是拿着地址偏移量直接到文件中取数据的，反观InnoDB是通过获取主键之后再去聚簇索引里边儿找记录，虽然说也不慢，但还是比不上直接用地址去访问。
> 此处我们只是非常简要的介绍了一下MyISAM的索引，具体细节全拿出来又可以写一篇文章了。这里只是希望大家理解InnoDB中的索引即数据，数据即索引，而MyISAM中却是索引是索引、数据是数据。

6.2.5 MySQL中创建和删除索引的语句

光顾着唠叨索引的原理了，那我们如何使用 MySQL 语句去建立这种索引呢？ InnoDB 和 MyISAM 会自动为主键或者声明为 UNIQUE 的列去自动建立 B+ 树索引，但是如果我们想为其他的列建立索引就需要我们显式的去指明。为啥不自动为每个列都建立个索引呢？别忘了，每建立一个索引都会建立一棵 B+ 树，每插入一条记录都要维护各个记录、数据页的排序关系，这是很费性能和存储空间的。

我们可以在创建表的时候指定需要建立索引的单个列或者建立联合索引的多个列：

```
CREATE TALBE 表名 (
    各种列的信息 ··· ,
    [KEY|INDEX] 索引名 (需要被索引的单个列或多个列)
)
```

其中的 KEY 和 INDEX 是同义词，任意选用一个就可以。我们也可以在修改表结构的时候添加索引：

```
ALTER TABLE 表名 ADD [INDEX|KEY] 索引名 (需要被索引的单个列或多个列);
```

也可以在修改表结构的时候删除索引：

```
ALTER TABLE 表名 DROP [INDEX|KEY] 索引名;
```

比方说我们想在创建 index_demo 表的时候就为 c2 和 c3 列添加一个 联合索引 ，可以这么写建表语句：

```
CREATE TABLE index_demo(
    c1 INT,
    c2 INT,
    c3 CHAR(1),
    PRIMARY KEY(c1),
    INDEX idx_c2_c3 (c2, c3)
);
```

在这个建表语句中我们创建的索引名是 idx_c2_c3 ，这个名称可以随便起，不过我们还是建议以 idx_ 为前缀，后边跟着需要建立索引的列名，多个列名之间用下划线 _ 分隔开。

如果我们想删除这个索引，可以这么写：

```
ALTER TABLE index_demo DROP INDEX idx_c2_c3;
```

7 第7章 好东西也得先学会怎么用-B+树索引的使用

标签： MySQL是怎样运行的

我们前边详细、详细又详细的唠叨了 InnoDB 存储引擎的 B+ 树索引，我们必须熟悉下边这些结论：

- 每个索引都对应一棵 B+ 树， B+ 树分为好多层，最下边一层是叶子节点，其余的是内节点。所有 用户记录 都存储在 B+ 树的叶子节点，所有 目录项记录 都存储在内节点。
- InnoDB 存储引擎会自动为主键（如果没有它会自动帮我们添加）建立 聚簇索引 ，聚簇索引的叶子节点包含完整的用户记录。
- 我们可以为自己感兴趣的列建立 二级索引 ， 二级索引 的叶子节点包含的用户记录由 索引列 + 主键 组成，所以如果想通过 二级索引 来查找完整的用户记录的话，需要通过 回表 操作，也就是在通过 二级索引 找到主键值之后再到 聚簇索引 中查找完整的用户记录。

- B+ 树中每层节点都是按照索引列值从小到大的顺序排序而组成了双向链表，而且每个页内的记录（不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单链表。如果是 联合索引 的话，则页面和记录先按照 联合索引 前边的列排序，如果该列值相同，再按照 联合索引 后边的列排序。
- 通过索引查找记录是从 B+ 树的根节点开始，一层一层向下搜索。由于每个页面都按照索引列的值建立了 Page Directory （页目录），所以在这些页面中的查找非常快。

如果你读上边的几点结论有些任何一点点疑惑的话，那下边的内容不适合你，回过头先去看前边的内容去。

7.1 索引的代价

在熟悉了 B+ 树索引原理之后，本篇文章的主题是唠叨如何更好的使用索引，虽然索引是个好东西，可不能乱建，在介绍如何更好的使用索引之前先要了解一下使用这玩意儿的代价，它在空间和时间上都会拖后腿：

- 空间上的代价

  这个是显而易见的，每建立一个索引都要为它建立一棵 B+ 树，每一棵 B+ 树的每一个节点都是一个数据页，一个页默认会占用 16KB 的存储空间，一棵很大的 B+ 树由许多数据页组成，那可是很大的一片存储空间呢。
- 时间上的代价

  每次对表中的数据进行增、删、改操作时，都需要去修改各个 B+ 树索引。而且我们讲过， B+ 树每层节点都是按照索引列的值从小到大的顺序排序而组成了双向链表。不论是叶子节点中的记录，还是内节点中的记录（也就是不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单向链表。而增、删、改操作可能会对节点和记录的排序造成破坏，所以存储引擎需要额外的时间进行一些记录移位，页面分裂、页面回收啥的操作来维护好节点和记录的排序。如果我们建了许多索引，每个索引对应的 B+ 树都要进行相关的维护操作，这还能不给性能拖后腿么？

所以说，一个表上索引建的越多，就会占用越多的存储空间，在增删改记录的时候性能就越差。为了能建立又好又少的索引，我们先得学学这些索引在哪些条件下起作用的。

7.2 B+树索引适用的条件

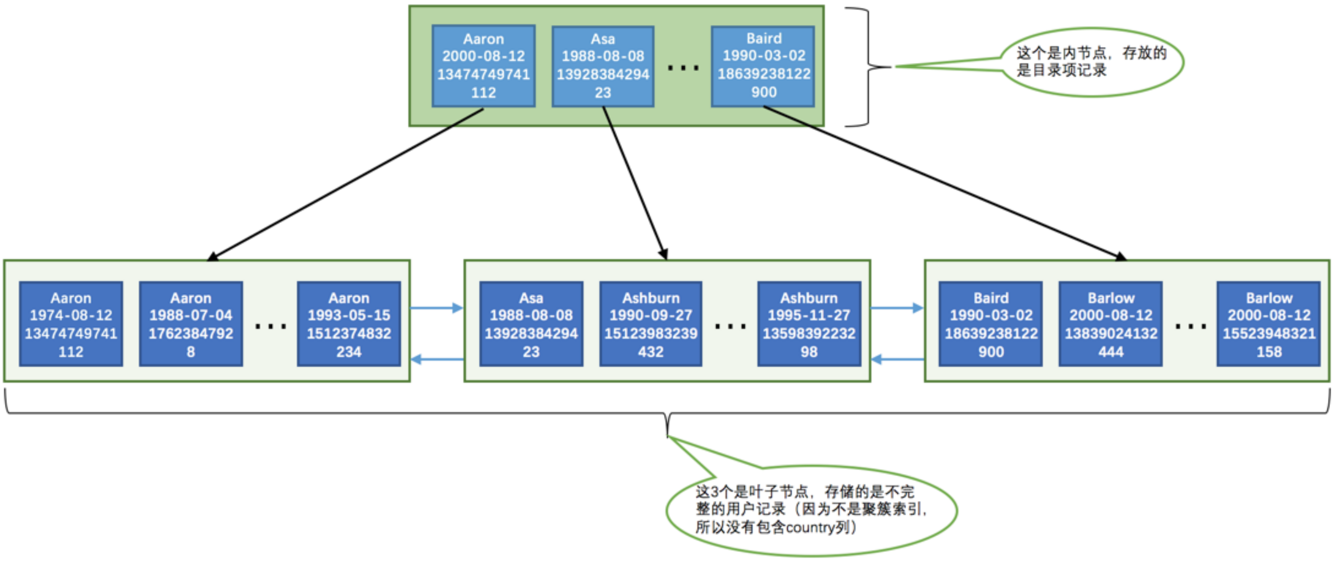
下边我们将唠叨许多种让 B+ 树索引发挥最大效能的技巧和注意事项，不过大家要清楚，所有的技巧都是源自你对 B+ 树索引本质的理解，所以如果你还不能保证对 B+ 树索引充分的理解，那么再次建议回过头把前边的内容看完了再来，要不然读文章对你来说是一种折磨。首先， B+ 树索引并不是万能的，并不是所有的查询语句都能用到我们建立的索引。下边介绍几个我们可能使用 B+ 树索引来进行查询的情况。为了故事的顺利发展，我们需要先创建一个表，这个表是用来存储人的一些基本信息的：

```
CREATE TABLE person_info(
    id INT NOT NULL auto_increment,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_birthday_phone_number (name, birthday, phone_number)
);
```

对于这个 person_info 表我们需要注意两点：

- 表中的主键是 id 列，它存储一个自动递增的整数。所以 InnoDB 存储引擎会自动为 id 列建立聚簇索引。
- 我们额外定义了一个二级索引 idx_name_birthday_phone_number ，它是由3个列组成的联合索引。所以在这个索引对应的 B+ 树的叶子节点处存储的用户记录只保留 name 、 birthday 、 phone_number 这三个列的值以及主键 id 的值，并不会保存 country 列的值。

从这两点注意中我们可以再次看到，一个表中有多少索引就会建立多少棵 B+ 树， person_info 表会为聚簇索引和 idx_name_birthday_phone_number 索引建立2棵 B+ 树。下边我们画一下索引 idx_name_birthday_phone_number 的示意图，不过既然我们已经掌握了 InnoDB 的 B+ 树索引原理，那我们在画图的时候为了让图更加清晰，所以在省略一些不必要的部分，比如记录的额外信息，各页面的页号等等，其中内节点中目录项记录的页号信息我们用箭头来代替，在记录结构中只保留 name 、 birthday 、 phone_number 、 id 这四个列的真实数据值，所以示意图就长这样（留心的同学看出来了，这其实和《高性能MySQL》里举的例子的图差不多，我觉得这个例子特别好，所以就借鉴了一下）：



为了方便大家理解，我们特意标明了哪些是内节点，哪些是叶子节点。再次强调一下，内节点中存储的是 目录项记录 ，叶子节点中存储的是 用户记录 （由于不是聚簇索引，所以用户记录是不完整的，缺少 country 列的值）。从图中可以看出，这个 idx_name_birthday_phone_number 索引对应的 B+ 树中页面和记录的排序方式就是这样的：

- 先按照 name 列的值进行排序。
- 如果 name 列的值相同，则按照 birthday 列的值进行排序。
- 如果 birthday 列的值也相同，则按照 phone_number 的值进行排序。

这个排序方式十分、特别、非常、巨、very very very重要，因为只要页面和记录是排好序的，我们就可以通过二分法来快速定位查找。下边的内容都仰仗这个图了，大家对照着图理解。

7.2.1 全值匹配

如果我们的搜索条件中的列和索引列一致的话，这种情况就称为全值匹配，比方说下边这个查找语句：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1990-09-27' AND phone_num
ber = '15123983239';
```

我们建立的 idx_name_birthday_phone_number 索引包含的3个列在这个查询语句中都展现出来了。大家可以想象一下这个查询过程：

- 因为 B+ 树的数据页和记录先是按照 name 列的值进行排序的，所以先可以很快定位 name 列的值是 Ashburn 的记录位置。
- 在 name 列相同的记录里又是按照 birthday 列的值进行排序的，所以在 name 列的值是 Ashburn 的记录里又可以快速定位 birthday 列的值是 '1990-09-27' 的记录。
- 如果很不幸， name 和 birthday 列的值都是相同的，那记录是按照 phone_number 列的值排序的，所以联合索引中的三个列都可能被用到。

有的同学也许有个疑问， WHERE 子句中的几个搜索条件的顺序对查询结果有啥影响么？也就是说如果我们调换 name 、 birthday 、 phone_number 这几个搜索列的顺序对查询的执行过程有影响么？比方说写成下边这样：

```
SELECT * FROM person_info WHERE birthday = '1990-09-27' AND phone_number = '15123983239' A
ND name = 'Ashburn';
```

答案是：没影响哈。 MySQL 有一个叫查询优化器的东东，会分析这些搜索条件并且按照可以使用的索引中列的顺序来决定先使用哪个搜索条件，后使用哪个搜索条件。我们后边儿会有专门的章节来介绍查询优化器，敬请期待。

7.2.2 匹配左边的列

其实在我们的搜索语句中也可以不用包含全部联合索引中的列，只包含左边的就行，比方说下边的查询语句：

```
SELECT * FROM person_info WHERE name = 'Ashburn';
```

或者包含多个左边的列也行：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1990-09-27';
```

那为什么搜索条件中必须出现左边的列才可以使用到这个 B+ 树索引呢？比如下边的语句就用不到这个 B+ 树索引么？

```
SELECT * FROM person_info WHERE birthday = '1990-09-27';
```

是的，的确用不到，因为 B+ 树的数据页和记录先是按照 name 列的值排序的，在 name 列的值相同的情况下才使用 birthday 列进行排序，也就是说 name 列的值不同的记录中 birthday 的值可能是无序的。而现在你跳过 name 列直接根据 birthday 的值去查找，臣妾做不到呀～ 那如果我就想在只使用 birthday 的值去通过 B+ 树索引进行查找咋办呢？这好办，你再对 birthday 列建一个 B+ 树索引就行了，创建索引的语法不用我唠叨了吧。

但是需要特别注意的一点是，如果我们想使用联合索引中尽可能多的列，搜索条件中的各个列必须是联合索引中从最左边连续的列。比方说联合索引 idx_name_birthday_phone_number 中列的定义顺序是 name 、 birthday 、 phone_number ，如果我们的搜索条件中只有 name 和 phone_number ，而没有中间的 birthday ，比方说这样：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND phone_number = '15123983239';
```

这样只能用到 name 列的索引， birthday 和 phone_number 的索引就用不上了，因为 name 值相同的记录先按照 birthday 的值进行排序， birthday 值相同的记录才按照 phone_number 值进行排序。

7.2.3 匹配列前缀

我们前边说过为某个列建立索引的意思其实就是在对应的 B+ 树的记录中使用该列的值进行排序，比方说 person_info 表上建立的联合索引 idx_name_birthday_phone_number 会先用 name 列的值进行排序，所以这个联合索引对应的 B+ 树中的记录的 name 列的排列就是这样的：

```
Aaron
Aaron
...
Aaron
Asa
Ashburn
...
Ashburn
Baird
Barlow
...
Barlow
```

字符串排序的本质就是比较哪个字符串大一点儿，哪个字符串小一点，比较字符串大小就用到了该列的字符集和比较规则，这个我们前边儿唠叨过，就不多唠叨了。这里需要注意的是，一般的比较规则都是逐个比较字符的大小，也就是说我们比较两个字符串的大小的过程其实是这样的：

- 先比较字符串的第一个字符，第一个字符小的那个字符串就比较小。
- 如果两个字符串的第一个字符相同，那就再比较第二个字符，第二个字符比较小的那个字符串就比较小。
- 如果两个字符串的第二个字符也相同，那就接着比较第三个字符，依此类推。

所以一个排好序的字符串列其实有这样的特点：

- 先按照字符串的第一个字符进行排序。
- 如果第一个字符相同再按照第二个字符进行排序。
- 如果第二个字符相同再按照第三个字符进行排序，依此类推。

也就是说这些字符串的前n个字符，也就是前缀都是排好序的，所以对于字符串类型的索引列来说，我们只匹配它的前缀也是可以快速定位记录的，比方说我们想查询名字以 `'As'` 开头的记录，那就可以这么写查询语句：

```
SELECT * FROM person_info WHERE name LIKE 'As%';
```

但是需要注意的是，如果只给出后缀或者中间的某个字符串，比如这样：

```
SELECT * FROM person_info WHERE name LIKE '%As%';
```

`MySQL` 就无法快速定位记录位置了，因为字符串中间有 `'As'` 的字符串并没有排好序，所以只能全表扫描了。有时候我们有一些匹配某些字符串后缀的需求，比方说某个表有一个 `url` 列，该列中存储了许多url：

```
+----------------+
| url            |
+----------------+
| www.baidu.com  |
| www.google.com |
| www.gov.cn     |
| ...            |
| www.wto.org    |
+----------------+
```

假设已经对该 `url` 列创建了索引，如果我们想查询以 `com` 为后缀的网址的话可以这样写查询条件： `WHERE url LIKE '%com'` ，但是这样的话无法使用该 `url` 列的索引。为了在查询时用到这个索引而不至于全表扫描，我们可以把后缀查询改写成前缀查询，不过我们就得把表中的数据全部逆序存储一下，也就是说我们可以这样保存 `url` 列中的数据：

```
+----------------+
| url            |
+----------------+
| moc.udiab.www  |
| moc.elgoog.www |
| nc.vog.www     |
| ...            |
| gro.otw.www    |
+----------------+
```

这样再查找以 com 为后缀的网址时搜索条件便可以这么写： WHERE url LIKE 'moc%' ，这样就可以用到索引了。

7.2.4 匹配范围值

回头看我们 idx_name_birthday_phone_number 索引的 B+ 树示意图，所有记录都是按照索引列的值从小到大的顺序排好序的，所以这极大的方便我们查找索引列的值在某个范围内的记录。比方说下边这个查询语句：

```
SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow';
```

由于 B+ 树中的数据页和记录是先按 name 列排序的，所以我们上边的查询过程其实是这样的：

- 找到 name 值为 Asa 的记录。
- 找到 name 值为 Barlow 的记录。
- 哦啦，由于所有记录都是由链表连起来的（记录之间用单链表，数据页之间用双链表），所以他们之间的记录都可以很容易的取出来喽～
- 找到这些记录的主键值，再到 聚簇索引 中 回表 查找完整的记录。

不过在使用联合进行范围查找的时候需要注意，如果对多个列同时进行范围查找的话，只有对索引最左边的那个列进行范围查找的时候才能用到 B+ 树索引，比方说这样：

```
SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow' AND birthday > '1980-01-01';
```

上边这个查询可以分成两个部分：

1. 通过条件 name > 'Asa' AND name < 'Barlow' 来对 name 进行范围，查找的结果可能有多条 name 值不同的记录，
2. 对这些 name 值不同的记录继续通过 birthday > '1980-01-01' 条件继续过滤。

这样子对于联合索引 idx_name_birthday_phone_number 来说，只能用到 name 列的部分，而用不到 birthday 列的部分，因为只有 name 值相同的情况下才能用 birthday 列的值进行排序，而这个查询中通过 name 进行范围查找的记录中可能并不是按照 birthday 列进行排序的，所以在搜索条件中继续以 birthday 列进行查找时是用不到这个 B+ 树索引的。

7.2.5 精确匹配某一列并范围匹配另外一列

对于同一个联合索引来说，虽然对多个列都进行范围查找时只能用到最左边那个索引列，但是如果左边的列是精确查找，则右边的列可以进行范围查找，比方说这样：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday > '1980-01-01' AND birthday < '2000-12-31' AND phone_number > '15100000000';
```

这个查询的条件可以分为3个部分：

1. name = 'Ashburn' ，对 name 列进行精确查找，当然可以使用 B+ 树索引了。

2. birthday > '1980-01-01' AND birthday < '2000-12-31' ，由于 name 列是精确查找，所以通过 name = 'Ashburn' 条件查找后得到的结果的 name 值都是相同的，它们会再按照 birthday 的值进行排序。所以此时对 birthday 列进行范围查找是可以用到 B+ 树索引的。
3. phone_number > '15100000000' ，通过 birthday 的范围查找的记录的 birthday 的值可能不同，所以这个条件无法再利用 B+ 树索引了，只能遍历上一步查询得到的记录。

同理，下边的查询也是可能用到这个 idx_name_birthday_phone_number 联合索引的：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1980-01-01' AND AND phone
_number > '15100000000';
```

7.2.6 用于排序

我们在写查询语句的时候经常需要对查询出来的记录通过 ORDER BY 子句按照某种规则进行排序。一般情况下，我们只能把记录都加载到内存中，再用一些排序算法，比如快速排序、归并排序、吧啦吧啦排序等等在内存中对这些记录进行排序，有的时候可能查询的结果集太大以至于不能在内存中进行排序的话，还可能暂时借助磁盘的空间来存放中间结果，排序操作完成后再把排好序的结果集返回到客户端。在 MySQL 中，把这种在内存中或者磁盘上进行排序的方式统称为文件排序（英文名： filesort ），跟 文件 这个词儿一沾边儿，就显得这些排序操作非常慢了（磁盘和内存的速度比起来，就像是飞机和蜗牛的对比）。但是如果 ORDER BY 子句里使用到了我们的索引列，就有可能省去在内存或文件中排序的步骤，比如下边这个简单的查询语句：

```
SELECT * FROM person_info ORDER BY name, birthday, phone_number LIMIT 10;
```

这个查询的结果集需要先按照 name 值排序，如果记录的 name 值相同，则需要按照 birthday 来排序，如果 birthday 的值相同，则需要按照 phone_number 排序。大家可以回过头去看我们建立的 idx_name_birthday_phone_number 索引的示意图，因为这个 B+ 树索引本身就是按照上述规则排好序的，所以直接从索引中提取数据，然后进行 回表 操作取出该索引中不包含的列就好了。简单吧？是的，索引就是这么牛逼。

7.2.6.1 使用联合索引进行排序注意事项

对于 联合索引 有个问题需要注意， ORDER BY 的子句后边的列的顺序也必须按照索引列的顺序给出，如果给出 ORDER BY phone_number, birthday, name 的顺序，那也是用不了 B+ 树索引，这种颠倒顺序就不能使用索引的原因我们上边详细说过了，这就不赘述了。

同理， ORDER BY name 、 ORDER BY name, birthday 这种匹配索引左边的列的形式可以使用部分的 B+ 树索引。当联合索引左边列的值为常量，也可以使用后边的列进行排序，比如这样：

```
SELECT * FROM person_info WHERE name = 'A' ORDER BY birthday, phone_number LIMIT 10;
```

这个查询能使用联合索引进行排序是因为 name 列的值相同的记录是按照 birthday , phone_number 排序的，说了好多遍了都。

7.2.6.2 不可以使用索引进行排序的几种情况

ASC、DESC混用

对于使用联合索引进行排序的场景，我们要求各个排序列的排序顺序是一致的，也就是要么各个列都是 ASC 规则排序，要么都是 DESC 规则排序。

> 小贴士：
> ORDER BY子句后的列如果不加ASC或者DESC默认是按照ASC排序规则排序的，也就是升序排序的。

为啥会有这种奇葩规定呢？这个还得回头想想这个 idx_name_birthday_phone_number 联合索引中记录的结构：

- 先按照记录的 name 列的值进行升序排列。
- 如果记录的 name 列的值相同，再按照 birthday 列的值进行升序排列。
- 如果记录的 birthday 列的值相同，再按照 phone_number 列的值进行升序排列。

如果查询中的各个排序列的排序顺序是一致的，比方说下边这两种情况：

- ORDER BY name, birthday LIMIT 10

  这种情况直接从索引的最左边开始往右读10行记录就可以了。
- ORDER BY name DESC, birthday DESC LIMIT 10 ，

  这种情况直接从索引的最右边开始往左读10行记录就可以了。

但是如果我们查询的需求是先按照 name 列进行升序排列，再按照 birthday 列进行降序排列的话，比如说这样的查询语句：

```
SELECT * FROM person_info ORDER BY name, birthday DESC LIMIT 10;
```

这样如果使用索引排序的话过程就是这样的：

- 先从索引的最左边确定 name 列最小的值，然后找到 name 列等于该值的所有记录，然后从 name 列等于该值的最右边的那条记录开始往左找10条记录。
- 如果 name 列等于最小的值的记录不足10条，再继续往右找 name 值第二小的记录，重复上边那个过程，直到找到10条记录为止。

累不累？累！重点是这样不能高效使用索引，而要采取更复杂的算法去从索引中取数据，设计 MySQL 的大叔觉得这样还不如直接文件排序来的快，所以就规定使用联合索引的各个排序列的排序顺序必须是一致的。

WHERE子句中出现非排序使用到的索引列

如果WHERE子句中出现了非排序使用到的索引列，那么排序依然是使用不到索引的，比方说这样：

```
SELECT * FROM person_info WHERE country = 'China' ORDER BY name LIMIT 10;
```

这个查询只能先把符合搜索条件 country = 'China' 的记录提取出来后再进行排序，是使用不到索引。注意和下边这个查询作区别：

```
SELECT * FROM person_info WHERE name = 'A' ORDER BY birthday, phone_number LIMIT 10;
```

虽然这个查询也有搜索条件，但是 name = 'A' 可以使用到索引 idx_name_birthday_phone_number ，而且过滤剩下的记录还是按照 birthday 、 phone_number 列排序的，所以还是可以使用索引进行排序的。

排序列包含非同一个索引的列

有时候用来排序的多个列不是一个索引里的，这种情况也不能使用索引进行排序，比方说：

```
SELECT * FROM person_info ORDER BY name, country LIMIT 10;
```

name 和 country 并不属于一个联合索引中的列，所以无法使用索引进行排序，至于为啥我就不想再唠叨了，自己用前边的理论自己捋一捋把~

排序列使用了复杂的表达式

要想使用索引进行排序操作，必须保证索引列是以单独列的形式出现，而不是修饰过的形式，比方说这样：

```
SELECT * FROM person_info ORDER BY UPPER(name) LIMIT 10;
```

使用了 UPPER 函数修饰过的列就不是单独的列啦，这样就无法使用索引进行排序啦。

7.2.7 用于分组

有时候我们为了方便统计表中的一些信息，会把表中的记录按照某些列进行分组。比如下边这个分组查询：

```
SELECT name, birthday, phone_number, COUNT(*) FROM person_info GROUP BY name, birthday, ph
one_number
```

这个查询语句相当于做了3次分组操作：

1. 先把记录按照 `name` 值进行分组，所有 `name` 值相同的记录划分为一组。
2. 将每个 `name` 值相同的分组里的记录再按照 `birthday` 的值进行分组，将 `birthday` 值相同的记录放到一个小分组里，所以看起来就像在一个大分组里又化分了好多小分组。
3. 再将上一步中产生的小分组按照 `phone_number` 的值分成更小的分组，所以整体上看起来就像是先把记录分成一个大分组，然后把 `大分组` 分成若干个 `小分组` ，然后把若干个 `小分组` 再细分成更多的 `小小分组` 。

然后针对那些 `小小分组` 进行统计，比如在我们这个查询语句中就是统计每个 `小小分组` 包含的记录条数。如果没有索引的话，这个分组过程全部需要在内存里实现，而如果有了索引的话，恰巧这个分组顺序又和我们的 `B+` 树中的索引列的顺序是一致的，而我们的 `B+` 树索引又是按照索引列排好序的，这不正好么，所以可以直接使用 `B+` 树索引进行分组。

和使用 `B+` 树索引进行排序是一个道理，分组列的顺序也需要和索引列的顺序一致，也可以只使用索引列中左边的列进行分组，吧啦吧啦的～

7.3 回表的代价

上边的讨论对 `回表` 这个词儿多是一带而过，可能大家没啥深刻的体会，下边我们详细唠叨下。还是用 `idx_name_birthday_phone_number` 索引为例，看下边这个查询：

```
SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow';
```

在使用 `idx_name_birthday_phone_number` 索引进行查询时大致可以分为这两个步骤：

1. 从索引 `idx_name_birthday_phone_number` 对应的 `B+` 树中取出 `name` 值在 `Asa` ～ `Barlow` 之间的用户记录。
2. 由于索引 `idx_name_birthday_phone_number` 对应的 `B+` 树用户记录中只包含 `name` 、 `birthday` 、 `phone_number` 、 `id` 这4个字段，而查询列表是 `*` ，意味着要查询表中所有字段，也就是还要包括 `country` 字段。这时需要把从上一步中获取到的每一条记录的 `id` 字段都到聚簇索引对应的 `B+` 树中找到完整的用户记录，也就是我们通常所说的 `回表` ，然后把完整的用户记录返回给查询用户。

由于索引 `idx_name_birthday_phone_number` 对应的 `B+` 树中的记录首先会按照 `name` 列的值进行排序，所以值在 `Asa` ～ `Barlow` 之间的记录在磁盘中的存储是相连的，集中分布在一个或几个数据页中，我们可以很快的把这些连着的记录从磁盘中读出来，这种读取方式我们也可以称为 `顺序I/O` 。根据第1步中获取到的记录的 `id` 字段的值可能并不相连，而在聚簇索引中记录是根据 `id` （也就是主键）的顺序排列的，所以根据这些并不连续的 `id` 值到聚簇索引中访问完整的用户记录可能分布在不同的数据页中，这样读取完整的用户记录可能要访问更多的数据页，这种读取方式我们也可以称为 `随机I/O` 。一般情况下，顺序I/O比随机I/O的性能高很多，所以步骤1的执行可能很快，而步骤2就慢一些。所以这个使用索引 `idx_name_birthday_phone_number` 的查询有这么两个特点：

- 会使用到两个 `B+` 树索引，一个二级索引，一个聚簇索引。
- 访问二级索引使用 `顺序I/O` ，访问聚簇索引使用 `随机I/O` 。

需要回表的记录越多，使用二级索引的性能就越低，甚至让某些查询宁愿使用全表扫描也不使用 `二级索引` 。比方说 `name` 值在 `Asa` ～ `Barlow` 之间的用户记录数量占全部记录数量90%以上，那么如果使用 `idx_name_birthday_phone_number` 索引的话，有90%多的 `id` 值需要回表，这不是吃力不讨好么，还不如直接去扫描聚簇索引（也就是全表扫描）。

那什么时候采用全表扫描的方式，什么时候使用采用 二级索引 + 回表 的方式去执行查询呢？这个就是传说中的查询优化器做的工作，查询优化器会事先对表中的记录计算一些统计数据，然后再利用这些统计数据根据查询的条件来计算一下需要回表的记录数，需要回表的记录数越多，就越倾向于使用全表扫描，反之倾向于使用 二级索引 + 回表 的方式。当然优化器做的分析工作不仅仅是这么简单，但是大致上是个这个过程。一般情况下，限制查询获取较少的记录数会让优化器更倾向于选择使用 二级索引 + 回表 的方式进行查询，因为回表的记录越少，性能提升就越高，比方说上边的查询可以改写成这样：

```
SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow' LIMIT 10;
```

添加了 LIMIT 10 的查询更容易让优化器采用 二级索引 + 回表 的方式进行查询。

对于有排序需求的查询，上边讨论的采用 全表扫描 还是 二级索引 + 回表 的方式进行查询的条件也是成立的，比方说下边这个查询：

```
SELECT * FROM person_info ORDER BY name, birthday, phone_number;
```

由于查询列表是 * ，所以如果使用二级索引进行排序的话，需要把排序完的二级索引记录全部进行回表操作，这样操作的成本还不如直接遍历聚簇索引然后再进行文件排序（ filesort ）低，所以优化器会倾向于使用 全表扫描 的方式执行查询。如果我们加了 LIMIT 子句，比如这样：

```
SELECT * FROM person_info ORDER BY name, birthday, phone_number LIMIT 10;
```

这样需要回表的记录特别少，优化器就会倾向于使用 二级索引 + 回表 的方式执行查询。

7.3.1 覆盖索引

为了彻底告别 回表 操作带来的性能损耗，我们建议：最好在查询列表里只包含索引列，比如这样：

```
SELECT name, birthday, phone_number FROM person_info WHERE name > 'Asa' AND name < 'Barlo
w'
```

因为我们只查询 name , birthday , phone_number 这三个索引列的值，所以在通过 idx_name_birthday_phone_number 索引得到结果后就不必到 聚簇索引 中再查找记录的剩余列，也就是 country 列的值了，这样就省去了 回表 操作带来的性能损耗。我们把这种只需要用到索引的查询方式称为 索引覆盖 。排序操作也优先使用 覆盖索引 的方式进行查询，比方说这个查询：

```
SELECT name, birthday, phone_number  FROM person_info ORDER BY name, birthday, phone_numbe
r;
```

虽然这个查询中没有 LIMIT 子句，但是采用了 覆盖索引 ，所以查询优化器就会直接使用 idx_name_birthday_phone_number 索引进行排序而不需要回表操作了。

当然，如果业务需要查询出索引以外的列，那还是以保证业务需求为重。但是我们很不鼓励用 * 号作为查询列表，最好把我们需要查询的列依次标明。

7.4 如何挑选索引

上边我们以 idx_name_birthday_phone_number 索引为例对索引的适用条件进行了详细的唠叨，下边看一下我们在建立索引时或者编写查询语句时就应该注意的一些事项。

7.4.1 只为用于搜索、排序或分组的列创建索引

也就是说，只为出现在 WHERE 子句中的列、连接子句中的连接列，或者出现在 ORDER BY 或 GROUP BY 子句中的列创建索引。而出现在查询列表中的列就没必要建立索引了：

```
SELECT birthday, country FROM person_name WHERE name = 'Ashburn';
```

像查询列表中的 birthday 、 country 这两个列就不需要建立索引，我们只需要为出现在 WHERE 子句中的 name 列创建索引就可以了。

7.4.2 考虑列的基数

列的基数 指的是某一列中不重复数据的个数，比方说某个列包含值 2, 5, 8, 2, 5, 8, 2, 5, 8 ，虽然有 9 条记录，但该列的基数却是 3 。也就是说，在记录行数一定的情况下，列的基数越大，该列中的值越分散，列的基数越小，该列中的值越集中。这个 列的基数 指标非常重要，直接影响我们是否能有效的利用索引。假设某个列的基数为 1 ，也就是所有记录在该列中的值都一样，那为该列建立索引是没有用的，因为所有值都一样就无法排序，无法进行快速查找了~ 而且如果某个建立了二级索引的列的重复值特别多，那么使用这个二级索引查出的记录还可能要做回表操作，这样性能损耗就更大了。所以结论就是：最好为那些列的基数大的列建立索引，为基数太小列的建立索引效果可能不好。

7.4.3 索引列的类型尽量小

我们在定义表结构的时候要显式的指定列的类型，以整数类型为例，有 TINYINT 、 MEDIUMINT 、 INT 、 BIGINT 这么几种，它们占用的存储空间依次递增，我们这里所说的 类型大小 指的就是该类型表示的数据范围的大小。能表示的整数范围当然也是依次递增，如果我们想要对某个整数列建立索引的话，在表示的整数范围允许的情况下，尽量让索引列使用较小的类型，比如我们能使用 INT 就不要使用 BIGINT ，能使用 MEDIUMINT 就不要使用 INT ~ 这是因为：

- 数据类型越小，在查询时进行的比较操作越快（这是CPU层次的东东）
- 数据类型越小，索引占用的存储空间就越少，在一个数据页内就可以放下更多的记录，从而减少磁盘 I/O 带来的性能损耗，也就意味着可以把更多的数据页缓存在内存中，从而加快读写效率。

这个建议对于表的主键来说更加适用，因为不仅是聚簇索引中会存储主键值，其他所有的二级索引的节点处都会存储一份记录的主键值，如果主键适用更小的数据类型，也就意味着节省更多的存储空间和更高效的 I/O 。

7.4.4 索引字符串值的前缀

我们知道一个字符串其实是由若干个字符组成，如果我们在 MySQL 中使用 utf8 字符集去存储字符串的话，编码一个字符需要占用 1~3 个字节。假设我们的字符串很长，那存储一个字符串就需要占用很大的存储空间。在我们需要为这个字符串列建立索引时，那就意味着在对应的 B+ 树中有这么两个问题：

- B+ 树索引中的记录需要把该列的完整字符串存储起来，而且字符串越长，在索引中占用的存储空间越大。
- 如果 B+ 树索引中索引列存储的字符串很长，那在做字符串比较时会占用更多的时间。

我们前边儿说过索引列的字符串前缀其实也是排好序的，所以索引的设计者提出了个方案 --- 只对字符串的前几个字符进行索引也就是说在二级索引的记录中只保留字符串前几个字符。这样在查找记录时虽然不能精确的定位到记录的位置，但是能定位到相应前缀所在的位置，然后根据前缀相同的记录的主键值回表查询完整的字符串值，再对比就好了。这样只在 B+ 树中存储字符串的前几个字符的编码，既节约空间，又减少了字符串的比较时间，还大概能解决排序的问题，何乐而不为，比方说我们在建表语句中只对 name 列的前10个字符进行索引可以这么写：

```
CREATE TABLE person_info(
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    KEY idx_name_birthday_phone_number (name(10), birthday, phone_number)
);
```

name(10) 就表示在建立的 B+ 树索引中只保留记录的前 10 个字符的编码，这种只索引字符串值的前缀的策略是我们非常鼓励的，尤其是在字符串类型能存储的字符比较多的时候。

7.4.4.1 索引列前缀对排序的影响

如果使用了索引列前缀，比方说前边只把 name 列的前10个字符放到了二级索引中，下边这个查询可能就有点儿尴尬了：

```
SELECT * FROM person_info ORDER BY name LIMIT 10;
```

因为二级索引中不包含完整的 name 列信息，所以无法对前十个字符相同，后边的字符不同的记录进行排序，也就是使用索引列前缀的方式无法支持使用索引排序，只好乖乖的用文件排序喽。

7.4.5 让索引列在比较表达式中单独出现

假设表中有一个整数列 my_col ，我们为这个列建立了索引。下边的两个 WHERE 子句虽然语义是一致的，但是在效率上却有差别：

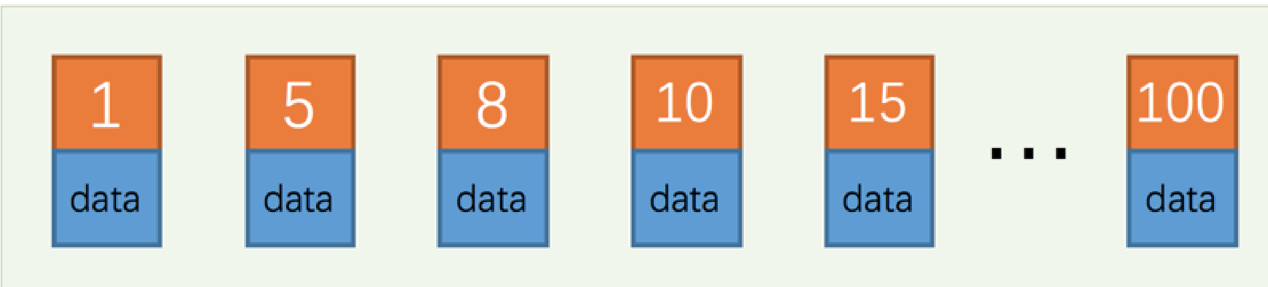
1. WHERE my_col * 2 < 4
2. WHERE my_col < 4/2

第1个 WHERE 子句中 my_col 列并不是以单独列的形式出现的，而是以 my_col * 2 这样的表达式的形式出现的，存储引擎会依次遍历所有的记录，计算这个表达式的值是不是小于 4 ，所以这种情况下是使用不到为 my_col 列建立的 B+ 树索引的。而第2个 WHERE 子句中 my_col 列并是以单独列的形式出现的，这样的情况可以直接使用 B+ 树索引。

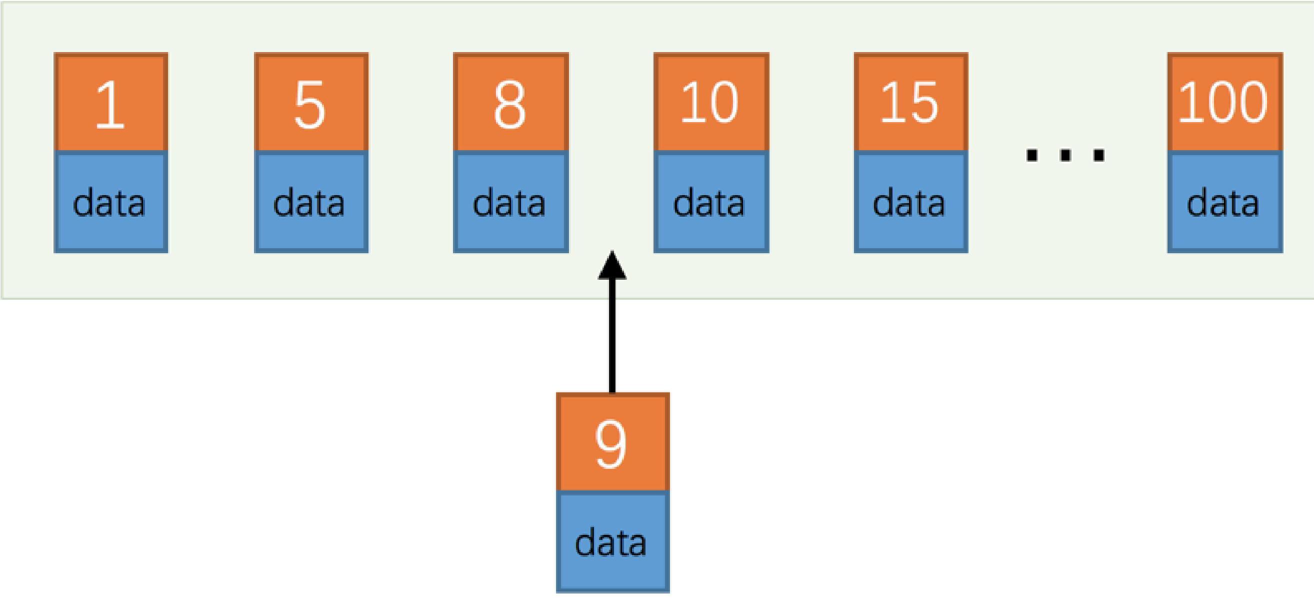
所以结论就是：如果索引列在比较表达式中不是以单独列的形式出现，而是以某个表达式，或者函数调用形式出现的话，是用不到索引的。

7.4.6 主键插入顺序

我们知道，对于一个使用 InnoDB 存储引擎的表来说，在我们没有显式的创建索引时，表中的数据实际上都是存储在 聚簇索引 的叶子节点的。而记录又是存储在数据页中的，数据页和记录又是按照记录主键值从小到大的顺序进行排序，所以如果我们插入的记录的主键值是依次增大的话，那我们每插满一个数据页就换到下一个数据页继续插，而如果我们插入的主键值忽大忽小的话，这就比较麻烦了，假设某个数据页存储的记录已经满了，它存储的主键值在 1~100 之间：



如果此时再插入一条主键值为 9 的记录，那它插入的位置就如下图：



可这个数据页已经满了啊，再插进来咋办呢？我们需要把当前页面分裂成两个页面，把本页中的一些记录移动到新创建的这个页中。页面分裂和记录移位意味着什么？意味着：性能损耗！所以如果我们想尽量避免这样无谓的性能损耗，最好让插入的记录的主键值依次递增，这样就不会发生这样的性能损耗了。所以我们建议：让主键具有 AUTO_INCREMENT ，让存储引擎自己为表生成主键，而不是我们手动插入 ，比方说我们可以这样定义 person_info 表：

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_birthday_phone_number (name(10), birthday, phone_number)
);
```

我们自定义的主键列 id 拥有 AUTO_INCREMENT 属性，在插入记录时存储引擎会自动为我们填入自增的主键值。

7.4.7 冗余和重复索引

有时候有的同学有意或者无意的就对同一个列创建了多个索引，比方说这样写建表语句：

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_birthday_phone_number (name(10), birthday, phone_number),
    KEY idx_name (name(10))
);
```

我们知道，通过 idx_name_birthday_phone_number 索引就可以对 name 列进行快速搜索，再创建一个专门针对 name 列的索引就算是一个 冗余 索引，维护这个索引只会增加维护的成本，并不会对搜索有什么好处。

另一种情况，我们可能会对某个列重复建立索引，比方说这样：

```
CREATE TABLE repeat_index_demo (
    c1 INT PRIMARY KEY,
    c2 INT,
    UNIQUE uidx_c1 (c1),
    INDEX idx_c1 (c1)
);
```

我们看到， c1 既是主键、又给它定义为一个唯一索引，还给它定义了一个普通索引，可是主键本身就会生成聚簇索引，所以定义的唯一索引和普通索引是重复的，这种情况要避免。

7.5 总结

上边只是我们在创建和使用 B+ 树索引的过程中需要注意的一些点，后边我们还会陆续介绍更多的优化方法和注意事项，敬请期待。本集内容总结如下：

1. B+ 树索引在空间和时间上都有代价，所以没事儿别瞎建索引。
2. B+ 树索引适用于下边这些情况：
    - 全值匹配
    - 匹配左边的列
    - 匹配范围值
    - 精确匹配某一列并范围匹配另外一列
    - 用于排序
    - 用于分组
3. 在使用索引时需要注意下边这些事项：
    - 只为用于搜索、排序或分组的列创建索引
    - 为列的基数大的列创建索引
    - 索引列的类型尽量小
    - 可以只对字符串值的前缀建立索引
    - 只有索引列在比较表达式中单独出现才可以适用索引
    - 为了尽可能少的让 聚簇索引 发生页面分裂和记录移位的情况，建议让主键拥有 AUTO_INCREMENT 属性。
    - 定位并删除表中的重复和冗余索引
    - 尽量使用 覆盖索引 进行查询，避免 回表 带来的性能损耗。

8 第8章 数据的家-MySQL的数据目录

标签： MySQL 是怎样运行的

8.1 数据库和文件系统的关系

我们知道像 InnoDB 、 MyISAM 这样的存储引擎都是把表存储在磁盘上的，而操作系统用来管理磁盘的那个东东又被称为 文件系统 ，所以用专业一点的话来表述就是：**像 *InnoDB* 、 *MyISAM* 这样的存储引擎都是把表存储在文件系统上的**。当我们想读取数据的时候，这些存储引擎会从文件系统中把数据读出来返回给我们，当我们想写入数据的时候，这些存储引擎会把这些数据又写回文件系统。本章就是要唠叨一下 InnoDB 和 MyISAM 这两个存储引擎的数据如何在文件系统中存储的。

8.2 MySQL数据目录

MySQL服务器程序在启动时会到文件系统的某个目录下加载一些文件，之后在运行过程中产生的数据也都会存储到这个目录下的某些文件中，这个目录就称为 数据目录 ，我们下边就要详细唠唠这个目录下具体都有哪些重要的东西。

8.2.1 数据目录和安装目录的区别

我们之前只接触过 MySQL 的安装目录（在安装 MySQL 的时候我们可以自己指定），我们重点强调过这个 安装目录 下非常重要的 bin 目录，它里边存储了许多关于控制客户端程序和服务器程序的命令（许多可执行文件，比如 mysql ， mysqld ， mysqld_safe 等等等等好几十个）。而 数据目录 是用来存储 MySQL 在运行过程中产生的数据，一定要和本章要讨论的 安装目录 区别开！一定要区分开！一定要区分开！一定要区分开！

8.2.2 如何确定MySQL中的数据目录

那说了半天，到底 MySQL 把数据都存到哪个路径下呢？其实 数据目录 对应着一个系统变量 datadir ，我们在使用客户端与服务器建立连接之后查看这个系统变量的值就可以了：

```
mysql> SHOW VARIABLES LIKE 'datadir';
+---------------+-----------------------+
| Variable_name | Value                 |
+---------------+-----------------------+
| datadir       | /usr/local/var/mysql/ |
+---------------+-----------------------+
1 row in set (0.00 sec)
```

从结果中可以看出，在我的计算机上 MySQL 的数据目录就是 /usr/local/var/mysql/ ，你用你的计算机试试呗~

8.3 数据目录的结构

MySQL 在运行过程中都会产生哪些数据呢？当然会包含我们创建的数据库、表、视图和触发器吧啦吧啦的用户数据，除了这些用户数据，为了程序更好的运行， MySQL 也会创建一些其他的额外数据，我们接下来细细的品味一下这个 数据目录 下的内容。

8.3.1 数据库在文件系统中的表示

每当我们使用 CREATE DATABASE 数据库名 语句创建一个数据库的时候，在文件系统上实际发生了什么呢？其实很简单，每个数据库都对应数据目录下的一个子目录，或者说对应一个文件夹，我们每当我们新建一个数据库时， MySQL 会帮我们做这两件事儿：

1. 在 数据目录 下创建一个和数据库名同名的子目录（或者说是文件夹）。
2. 在该与数据库名同名的子目录下创建一个名为 db.opt 的文件，这个文件中包含了该数据库的各种属性，比方说该数据库的字符集和比较规则是个啥。

比方说我们查看一下在我的计算机上当前有哪些数据库：

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| charset_demo_db    |
| dahaizi            |
| mysql              |
| performance_schema |
| sys                |
| xiaohaizi          |
+--------------------+
7 rows in set (0.00 sec)
```

可以看到在我的计算机上当前有7个数据库，其中 charset_demo_db 、 dahaizi 和 xiaohaizi 数据库是我们自定义的，其余4个数据库是属于MySQL自带的系统数据库。我们再看一下我的计算机上的 数据目录 下的内容：

```
.
├── auto.cnf
├── ca-key.pem
├── ca.pem
├── charset_demo_db
├── client-cert.pem
├── client-key.pem
├── dahaizi
├── ib_buffer_pool
├── ib_logfile0
├── ib_logfile1
├── ibdata1
├── ibtmp1
├── mysql
├── performance_schema
├── private_key.pem
├── public_key.pem
├── server-cert.pem
├── server-key.pem
├── sys
├── xiaohaizideMacBook-Pro.local.err
├── xiaohaizideMacBook-Pro.local.pid
└── xiaohaizi

6 directories, 16 files
```

当然这个数据目录下的文件和子目录比较多哈，但是如果仔细看的话，除了 information_schema 这个系统数据库外，其他的数据库在 数据目录 下都有对应的子目录。这个 information_schema 比较特殊，设计MySQL的大叔们对它的实现进行了特殊对待，没有使用相应的数据库目录，我们忽略它的存在就好了哈。

8.3.2 表在文件系统中的表示

我们的数据其实都是以记录的形式插入到表中的，每个表的信息其实可以分为两种：

1. 表结构的定义
2. 表中的数据

表结构 就是该表的名称是啥，表里边有多少列，每个列的数据类型是啥，有啥约束条件和索引，用的是啥字符集和比较规则吧啦吧啦的各种信息，这些信息都体现在了我们的建表语句中了。为了保存这些信息， InnoDB 和 MyISAM 这两种存储引擎都在 数据目录 下对应的数据库子目录下创建了一个专门用于描述表结构的文件，文件名是这样：

```
表名.frm
```

比方说我们在 dahaizi 数据库下创建一个名为 test 的表：

```
mysql> USE dahaizi;
Database changed

mysql> CREATE TABLE test (
    ->     c1 INT
    -> );
Query OK, 0 rows affected (0.03 sec)
```

那在数据库 dahaizi 对应的子目录下就会创建一个名为 test.frm 的用于描述表结构的文件。值得注意的是，这个后缀名为.frm是以二进制格式存储的，我们直接打开会是乱码的～ 你还不赶紧在你的计算机上创建个表试试～

描述表结构的文件我们知道怎么存储了，那表中的数据存到什么文件中了呢？在这个问题上，不同的存储引擎就产生了分歧了，下边我们分别看一下 InnoDB 和 MyISAM 是用什么文件来保存表中数据的。

8.3.2.1 InnoDB是如何存储表数据的

我们前边重点唠叨过 InnoDB 的一些实现原理，到现在为止我们应该熟悉下边这些东东：

- InnoDB 其实是使用 页 为基本单位来管理存储空间的，默认的 页 大小为 16KB 。
- 对于 InnoDB 存储引擎来说，每个索引都对应着一棵 B+ 树，该 B+ 树的每个节点都是一个数据页，数据页之间不必要是物理连续的，因为数据页之间有 双向链表 来维护着这些页的顺序。
- InnoDB 的聚簇索引的叶子节点存储了完整的用户记录，也就是所谓的索引即数据，数据即索引。

为了更好的管理这些页，设计 InnoDB 的大叔们提出了一个 表空间 或者 文件空间 （英文名： table space 或者 file space ）的概念，这个表空间是一个抽象的概念，它可以对应文件系统上一个或多个真实文件（不同表空间对应的文件数量可能不同）。每一个 表空间 可以被划分为很多很多很多个 页 ，我们的表数据就存放在某个 表空间 下的某些页里。设计 InnoDB 的大叔将表空间划分为几种不同的类型，我们一个一个看一下。

系统表空间（system tablespace）

这个所谓的 系统表空间 可以对应文件系统上一个或多个实际的文件，默认情况下， InnoDB 会在 数据目录 下创建一个名为 ibdata1 （在你的数据目录下找找看有木有）、大小为 12M 的文件，这个文件就是对应的 系统表空间 在文件系统上的表示。怎么才 12M ？这么点儿还没插多少数据就用完了，哈哈，那是因为这个文件是所谓的 自扩展文件 ，也就是当不够用的时候它会自己增加文件大小～

当然，如果你想让系统表空间对应文件系统上多个实际文件，或者仅仅觉得原来的 ibdata1 这个文件名难听，那可以在 MySQL 启动时配置对应的文件路径以及它们的大小，比如我们这样修改一下配置文件：

```
[server]
innodb_data_file_path=data1:512M;data2:512M:autoextend
```

这样在 MySQL 启动之后就会创建这两个512M大小的文件作为 系统表空间 ，其中的 autoextend 表明这两个文件如果不够用会自动扩展 data2 文件的大小。

我们也可以把 系统表空间 对应的文件路径不配置到 数据目录 下，甚至可以配置到单独的磁盘分区上，涉及到的启动参数就是 innodb_data_file_path 和 innodb_data_home_dir ，具体的配置逻辑挺绕的，我们这就不多唠叨了，知道改哪个参数可以修改 系统表空间 对应的文件，有需要的时候到官方文档里一查就好了。

需要注意的一点是，在一个MySQL服务器中，系统表空间只有一份。从MySQL5.5.7到MySQL5.6.6之间的各个版本中，我们表中的数据都会被默认存储到这个 ***系统表空间***。

独立表空间(file-per-table tablespace)

在MySQL5.6.6以及之后的版本中， InnoDB 并不会默认的把各个表的数据存储到系统表空间中，而是为每一个表建立一个独立表空间，也就是说我们创建了多少个表，就有多少个独立表空间。使用 独立表空间 来存储表数据的话，会在该表所属数据库对应的子目录下创建一个表示该 独立表空间 的文件，文件名和表名相同，只不过添加了一个 .ibd 的扩展名而已，所以完整的文件名称长这样：

```
表名.ibd
```

比方说假如我们使用了 独立表空间 去存储 xiaohaizi 数据库下的 test 表的话，那么在该表所在数据库对应的 xiaohaizi 目录下会为 test 表创建这两个文件：

```
test.frm
test.ibd
```

其中 test.ibd 文件就用来存储 test 表中的数据和索引。当然我们也可以自己指定使用 系统表空间 还是 独立表空间 来存储数据，这个功能由启动参数 innodb_file_per_table 控制，比如说我们想刻意将表数据都存储到 系统表空间 时，可以在启动 MySQL 服务器的时候这样配置：

```
[server]
innodb_file_per_table=0
```

当 innodb_file_per_table 的值为 0 时，代表使用系统表空间；当 innodb_file_per_table 的值为 1 时，代表使用独立表空间。不过 innodb_file_per_table 参数只对新建的表起作用，对于已经分配了表空间的表并不起作用。如果我们想把已经存在系统表空间中的表转移到独立表空间，可以使用下边的语法：

```
ALTER TABLE 表名 TABLESPACE [=] innodb_file_per_table;
```

或者把已经存在独立表空间的表转移到系统表空间，可以使用下边的语法：

```
ALTER TABLE 表名 TABLESPACE [=] innodb_system;
```

其中中括号扩起来的 = 可有可无，比方说我们想把 test 表从独立表空间移动到系统表空间，可以这么写：

```
ALTER TABLE test TABLESPACE innodb_system;
```

其他类型的表空间

随着MySQL的发展，除了上述两种老牌表空间之外，现在还新提出了一些不同类型的表空间，比如通用表空间（general tablespace）、undo表空间（undo tablespace）、临时表空间（temporary tablespace）吧啦吧啦的，具体情况我们就不细唠叨了，等用到的时候再提。

8.3.2.2 MyISAM是如何存储表数据的

好了，唠叨完了 InnoDB 的系统表空间和独立表空间，现在轮到 MyISAM 了。我们知道不像 InnoDB 的索引和数据是一个东东，在 MyISAM 中的索引全部都是 二级索引 ，该存储引擎的数据和索引是分开存放的。所以在文件系统中也是使用不同的文件来存储数据文件和索引文件。而且和 InnoDB 不同的是， MyISAM 并没有什么所谓的 表空间 一说，表数据都存放到对应的数据库子目录下。假如 test 表使用 MyISAM 存储引擎的话，那么在它所在数据库对应的 xiaohaizi 目录下会为 test 表创建这三个文件：

```
test.frm
test.MYD
test.MYI
```

其中 test.MYD 代表表的数据文件，也就是我们插入的用户记录； test.MYI 代表表的索引文件，我们为该表创建的索引都会放到这个文件中。

8.3.3 视图在文件系统中的表示

我们知道 MySQL 中的视图其实是虚拟的表，也就是某个查询语句的一个别名而已，所以在存储 视图 的时候是不需要存储真实的数据的，只需要把它的结构存储起来就行了。和 表 一样，描述视图结构的文件也会被存储到所属数据库对应的子目录下边，只会存储一个 视图名.frm 的文件。

8.3.4 其他的文件

除了我们上边说的这些用户自己存储的数据以外， 数据目录 下还包括为了更好运行程序的一些额外文件，主要包括这几种类型的文件：

- 服务器进程文件。

  我们知道每运行一个 MySQL 服务器程序，都意味着启动一个进程。 MySQL 服务器会把自己的进程ID写入到一个文件中。
- 服务器日志文件。

  在服务器运行过程中，会产生各种各样的日志，比如常规的查询日志、错误日志、二进制日志、redo日志吧啦吧啦各种日志，这些日志各有各的用途，我们之后会重点唠叨各种日志的用途，现在先了解一下就可以了。
- 默认/自动生成的SSL和RSA证书和密钥文件。

  主要是为了客户端和服务器安全通信而创建的一些文件， 大家看不懂可以忽略～

8.4 文件系统对数据库的影响

因为 MySQL 的数据都是存在文件系统中的，就不得不受到文件系统的一些制约，这在数据库和表的命名、表的大小和性能方面体现的比较明显，比如下边这些方面：

- 数据库名称和表名称不得超过文件系统所允许的最大长度。

  每个数据库都对应 数据目录 的一个子目录，数据库名称就是这个子目录的名称；每个表都会在数据库子目录下产生一个和表名同名的 .frm 文件，如果是 InnoDB 的独立表空间或者使用 MyISAM 引擎还会有别的文件的名称与表名一致。这些目录或文件名的长度都受限于文件系统所允许的长度～
- 特殊字符的问题

  为了避免因为数据库名和表名出现某些特殊字符而造成文件系统不支持的情况， MySQL 会把数据库名和表名中所有除数字和拉丁字母以外的所有字符在文件名里都映射成 @+编码值 的形式作为文件名。比方说我们创建的表的名称为 'test?' ，由于 ? 不属于数字或者拉丁字母，所以会被映射成编码值，所以这个表对应的 .frm 文件的名称就变成了 test@003f.frm 。
- 文件长度受文件系统最大长度限制

  对于 InnoDB 的独立表空间来说，每个表的数据都会被存储到一个与表名同名的 .ibd 文件中；对于 MyISAM 存储引擎来说，数据和索引会分别存放到与表同名的 .MYD 和 .MYI 文件中。这些文件会随着表中记录的增加而增大，它们的大小受限于文件系统支持的最大文件大小。

8.5 MySQL系统数据库简介

我们前边提到了MySQL的几个系统数据库，这几个数据库包含了MySQL服务器运行过程中所需的一些信息以及一些运行状态信息，我们现在稍微了解一下。

- mysql

  这个数据库贼核心，它存储了MySQL的用户账户和权限信息，一些存储过程、事件的定义信息，一些运行过程中产生的日志信息，一些帮助信息以及时区信息等。
- information_schema

  这个数据库保存着MySQL服务器维护的所有其他数据库的信息，比如有哪些表、哪些视图、哪些触发器、哪些列、哪些索引吧啦吧啦。这些信息并不是真实的用户数据，而是一些描述性信息，有时候也称之为元数据。
- performance_schema

  这个数据库里主要保存MySQL服务器运行过程中的一些状态信息，算是对MySQL服务器的一个性能监控。包括统计最近执行了哪些语句，在执行过程的每个阶段都花费了多长时间，内存的使用情况等等信息。
- sys

  这个数据库主要是通过视图的形式把 information_schema 和 performance_schema 结合起来，让程序员可以更方便的了解MySQL服务器的一些性能信息。

啥？这四个系统数据库这就介绍完了？是的，我们的标题写的就是 简介 嘛！如果真的要唠叨一下这几个系统库的使用，那怕是又要写一本书了... 这里只是因为介绍数据目录里遇到了，为了内容的完整性跟大家提一下，具体如何使用还是要参照文档~

9 第9章 存放页面的大池子-InnoDB的表空间

标签： MySQL是怎样运行的

通过前边儿的内容大家知道， 表空间 是一个抽象的概念，对于系统表空间来说，对应着文件系统中一个或多个实际文件；对于每个独立表空间来说，对应着文件系统中一个名为 表名.ibd 的实际文件。大家可以把表空间想象成被切分为许许多多个 页 的池子，当我们想为某个表插入一条记录的时候，就从池子中捞出一个对应的页来把数据写进去。本章内容会深入到表空间的各个细节中，带领大家在 InnoDB 存储结构的池子中畅游。由于本章中将会涉及比较多的概念，虽然这些概念都不难，但是却相互依赖，所以奉劝大家在看的时候：

- 不要跳着看！
- 不要跳着看！
- 不要跳着看！

9.1 回忆一些旧知识

9.1.1 页面类型

再一次强调，InnoDB是以页为单位管理存储空间的，我们的聚簇索引（也就是完整的表数据）和其他的二级索引都是以 B+ 树的形式保存到表空间的，而 B+ 树的节点就是数据页。我们前边说过，这个数据页的类型名其实是： FIL_PAGE_INDEX ，除了这种存放索引数据的页面类型之外，InnoDB也为了不同的目的设计了若干种不同类型的页面，为了唤醒大家的记忆，我们再一次把各种常用的页面类型提出来：

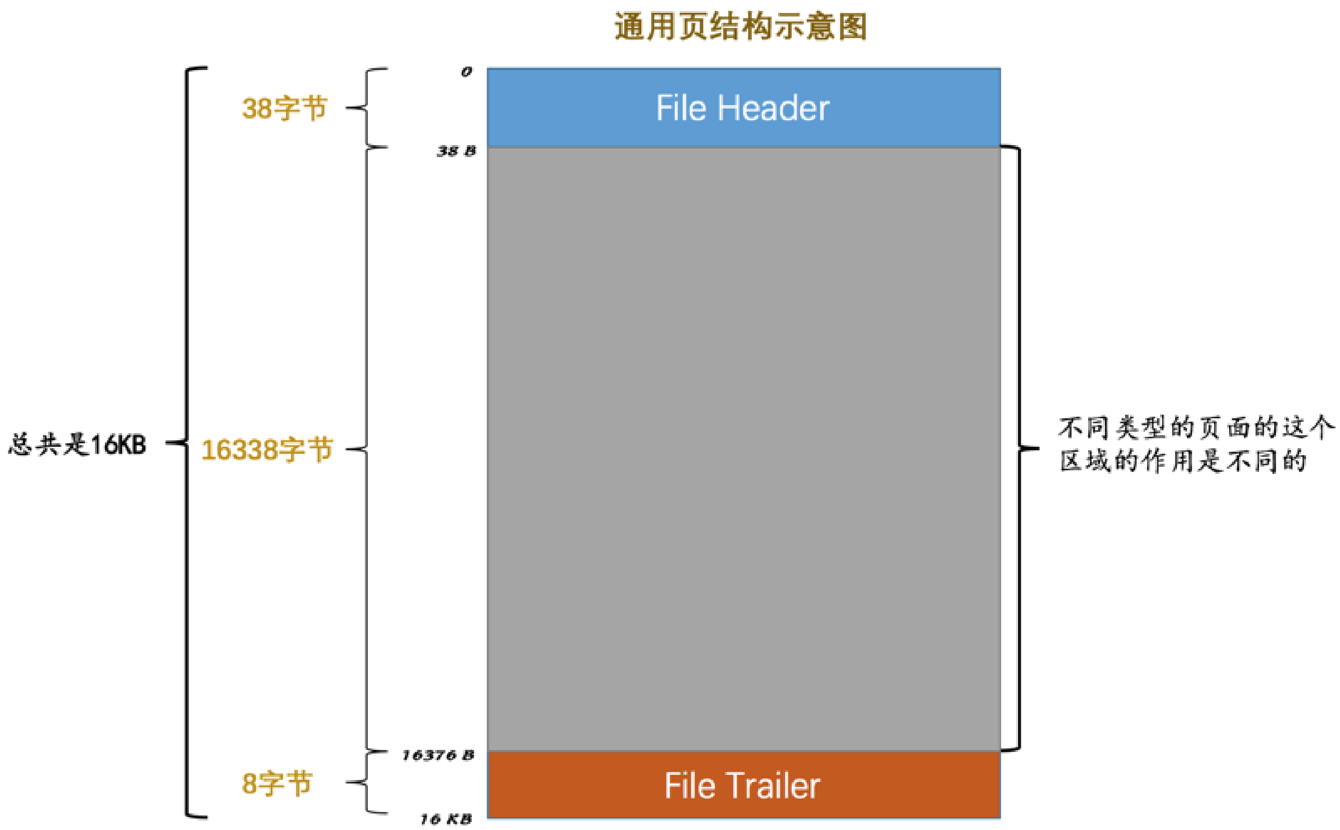
类型名称	十六进制	描述
FIL_PAGE_TYPE_ALLOCATED	0x0000	最新分配，还没使用
FIL_PAGE_UNDO_LOG	0x0002	Undo日志页
FIL_PAGE_INODE	0x0003	段信息节点
FIL_PAGE_IBUF_FREE_LIST	0x0004	Insert Buffer空闲列表

类型名称	十六进制	描述
FIL_PAGE_IBUF_BITMAP	0x0005	Insert Buffer位图
FIL_PAGE_TYPE_SYS	0x0006	系统页
FIL_PAGE_TYPE_TRX_SYS	0x0007	事务系统数据
FIL_PAGE_TYPE_FSP_HDR	0x0008	表空间头部信息
FIL_PAGE_TYPE_XDES	0x0009	扩展描述页
FIL_PAGE_TYPE_BLOB	0x000A	BLOB页
FIL_PAGE_INDEX	0x45BF	索引页，也就是我们所说的 数据页

因为页面类型前边都有个 FIL_PAGE 或者 FIL_PAGE_TYPE 的前缀，为简便起见我们后边唠叨页面类型的时候就把这些前缀省略掉了，比方说 FIL_PAGE_TYPE_ALLOCATED 类型称为 ALLOCATED 类型， FIL_PAGE_INDEX 类型称为 INDEX 类型。

9.1.2 页面通用部分

我们前边说过数据页，也就是 INDEX 类型的页由7个部分组成，其中的两个部分是所有类型的页面都通用的。当然我不能寄希望于你把我说的话都记住，所以在这里重新强调一遍，任何类型的页面都有下边这种通用的结构：



从上图中可以看出，任何类型的页都会包含这两个部分：

- File Header ：记录页面的一些通用信息
- File Trailer ：校验页是否完整，保证从内存到磁盘刷新时内容的一致性。

对于 File Trailer 我们不再做过多强调，全部忘记了的话可以到将数据页的那一章回顾一下。我们这里再强调一遍 File Header 的各个组成部分：

名称	占用空间大小	描述
FIL_PAGE_SPACE_OR_CHKSUM	4 字节	页的校验和（checksum值）
FIL_PAGE_OFFSET	4 字节	页号
FIL_PAGE_PREV	4 字节	上一个页的页号
FIL_PAGE_NEXT	4 字节	下一个页的页号
FIL_PAGE_LSN	8 字节	页面被最后修改时对应的日志序列位置（英文名是：Log Sequence Number）
FIL_PAGE_TYPE	2 字节	该页的类型
FIL_PAGE_FILE_FLUSH_LSN	8 字节	仅在系统表空间的一个页中定义，代表文件至少被刷新到了对应的LSN值
FIL_PAGE_ARCH_LOG_NO_OR_SPACE_ID	4 字节	页属于哪个表空间

现在除了名称里边儿带有 LSN 的两个字段大家可能看不懂以外，其他的字段肯定都是倍儿熟了，不过我们仍要强调这么几点：

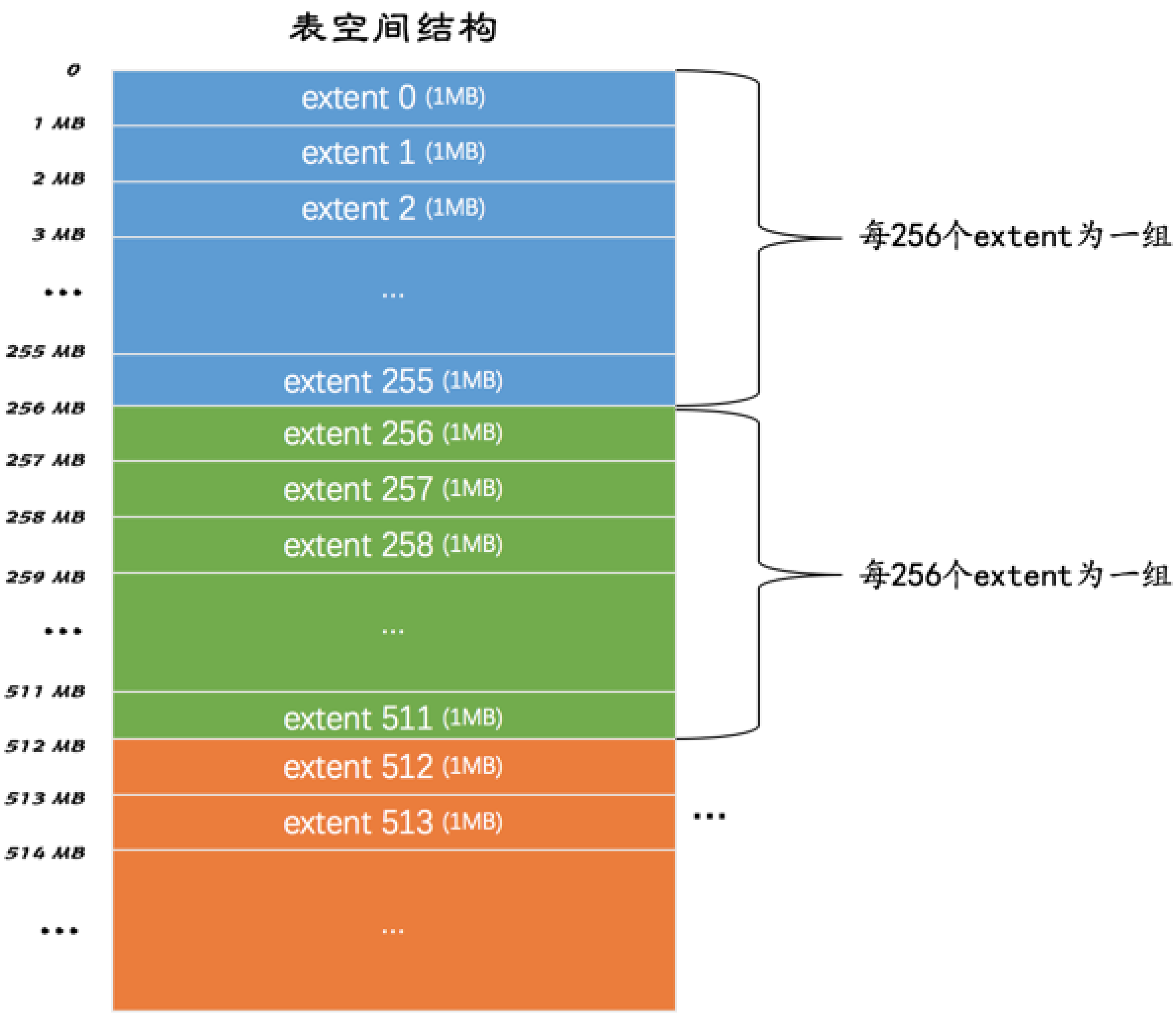
- 表空间中的每一个页都对应着一个页号，也就是 FIL_PAGE_OFFSET ，这个页号由4个字节组成，也就是32个比特位，所以一个表空间最多可以拥有2^{32}个页，如果按照页的默认大小16KB来算，一个表空间最多支持64TB的数据。表空间的第一个页的页号为0，之后的页号分别是1，2，3...依此类推
- 某些类型的页可以组成链表，链表中的页可以不按照物理顺序存储，而是根据 FIL_PAGE_PREV 和 FIL_PAGE_NEXT 来存储上一个页和下一个页的页号。需要注意的是，这两个字段主要是为了 INDEX 类型的页，也就是我们之前一直说的数据页建立 B+ 树后，为每层节点建立双向链表用的，一般类型的页是不使用这两个字段的。
- 每个页的类型由 FIL_PAGE_TYPE 表示，比如像数据页的该字段的值就是 0x45BF ，我们后边会介绍各种不同类型的页，不同类型的页在该字段上的值是不同的。

9.2 独立表空间结构

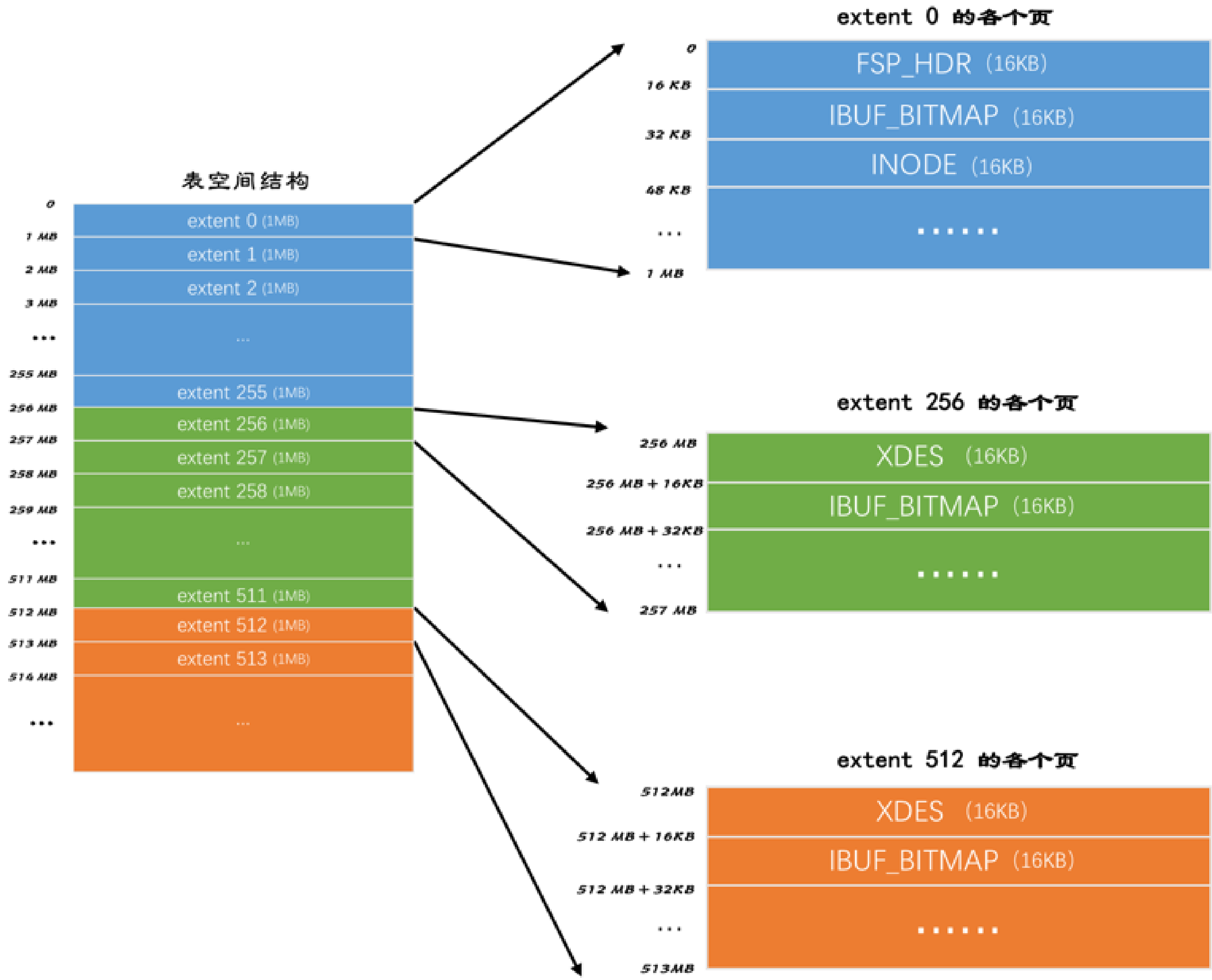
我们知道 InnoDB 支持许多种类型的表空间，本章重点关注独立表空间和系统表空间的结构。它们的结构比较相似，但是由于系统表空间中额外包含了一些关于整个系统的信息，所以我们先挑简单一点的独立表空间来唠叨，稍后再说系统表空间的结构。

9.2.1 区（extent）的概念

表空间中的页实在是太多了，为了更好的管理这些页面，设计 InnoDB 的大叔们提出了 区 （英文名： extent ）的概念。对于16KB的页来说，连续的64个页就是一个 区 ，也就是说一个区默认占用1MB空间大小。不论是系统表空间还是独立表空间，都可以看成是由若干个区组成的，每256个区被划分成一组。画个图表示就是这样：



其中 extent 0 ~ extent 255 这256个区算是第一个组， extent 256 ~ extent 511 这256个区算是第二个组， extent 512 ~ extent 767 这256个区算是第三个组（上图中并未画全第三个组全部的区，请自行脑补），依此类推可以划分更多的组。这些组的头几个页面的类型都是类似的，就像这样：



从上图中我们能得到如下信息：

- 第一个组最开始的3个页面的类型是固定的，也就是说 extent 0 这个区最开始的3个页面的类型是固定的，分别是：
  - FSP_HDR 类型：这个类型的页面是用来登记整个表空间的一些整体属性以及本组所有的 区 ，也就是 extent 0 ~ extent 255 这256个区的属性，稍后详细唠叨。需要注意的一点是，整个表空间只有一个 FSP_HDR 类型的页面。
  - IBUF_BITMAP 类型：这个类型的页面是存储本组所有的区的所有页面关于 INSERT BUFFER 的信息。当然，你现在不用知道啥是个 INSERT BUFFER ，后边会详细说到你吐。
  - INODE 类型：这个类型的页面存储了许多称为 INODE 的数据结构，还是那句话，现在你不需要知道啥是个 INODE ，后边儿会说到你吐。
- 其余各组最开始的2个页面的类型是固定的，也就是说 extent 256 、 extent 512 这些区最开始的2个页面的类型是固定的，分别是：
  - XDES 类型：全称是 extent descriptor ，用来登记本组256个区的属性，也就是说对于在 extent 256 区中的该类型页面存储的就是 extent 256 ~ extent 511 这些区的属性，对于在 extent 512 区中的该类型页面存储的就是 extent 512 ~ extent 767 这些区的属性。上边介绍的 FSP_HDR 类型的页面其实和 XDES 类型的页面的作用类似，只不过 FSP_HDR 类型的页面还会额外存储一些表空间的属性。
  - IBUF_BITMAP 类型：上边介绍过了。

好了，宏观的结构介绍完了，里边儿的名词大家也不用记清楚，只要大致记得：表空间被划分为许多连续的 区 ，每个区默认由64个页组成，每256个区划分为一组，每个组的最开始的几个页面类型是固定的就好了。

9.2.2 段（segment）的概念

为啥好端端的提出一个 区 （ extent ）的概念呢？我们以前分析问题的套路都是这样的：表中的记录存储到页里边儿，然后页作为节点组成 B+ 树，这个 B+ 树就是索引，然后吧啦吧啦一堆聚簇索引和二级索引的区别。这套路也没啥不妥的呀～

是的，如果我们表中数据量很少的话，比如说你的表中只有几十条、几百条数据的话，的确用不到 区 的概念，因为简单的几个页就能把对应的数据存储起来，但是你架不住表里的记录越来越多呀。

？？啥？？表里的记录多了又怎样？ B+ 树的每一层中的页都会形成一个双向链表呀， File Header 中的 FIL_PAGE_PREV 和 FIL_PAGE_NEXT 字段不就是为了形成双向链表设置的么？

是的是的，您说的都对，从理论上说，不引入 区 的概念只使用 页 的概念对存储引擎的运行并没啥影响，但是我们来考虑一下下边这个场景：

- 我们每向表中插入一条记录，本质上就是向该表的聚簇索引以及所有二级索引代表的 B+ 树的节点中插入数据。而 B+ 树的每一层中的页都会形成一个双向链表，如果是以 页 为单位来分配存储空间的话，双向链表相邻的两个页之间的物理位置可能离得非常远。我们介绍 B+ 树索引的适用场景的时候特别提到范围查询只需要定位到最左边的记录和最右边的记录，然后沿着双向链表一直扫描就可以了，而如果链表中相邻的两个页物理位置离得非常远，就是所谓的 随机I/O 。再一次强调，磁盘的速度和内存的速度差了好几个数量级， 随机I/O 是非常慢的，所以我们应该尽量让链表中相邻的页的物理位置也相邻，这样进行范围查询的时候才可以使用所谓的 顺序I/O 。

所以，所以，所以才引入了 区 （ extent ）的概念，一个区就是在物理位置上连续的64个页。在表中数据量大的时候，为某个索引分配空间的时候就不再按照页为单位分配了，而是按照 区 为单位分配，甚至在表中的数据十分非常特别多的时候，可以一次性分配多个连续的区。虽然可能造成一点点空间的浪费（数据不足填充满整个区），但是从性能角度看，可以消除很多的随机 I/O ，功大于过嘛！

事情到这里就结束了么？太天真了，我们提到的范围查询，其实是对 B+ 树叶子节点中的记录进行顺序扫描，而如果不区分叶子节点和非叶子节点，统统把节点代表的页面放到申请到的区中的话，进行范围扫描的效果就大打折扣了。所以设计 InnoDB 的大叔们对 B+ 树的叶子节点和非叶子节点进行了区别对待，也就是说叶子节点有自己独有的 区 ，非叶子节点也有自己独有的 区 。存放叶子节点的区的集合就算是一个 段 （ segment ），存放非叶子节点的区的集合也算是一个 段 。也就是说一个索引会生成2个段，一个叶子节点段，一个非叶子节点段。

默认情况下一个使用 InnoDB 存储引擎的表只有一个聚簇索引，一个索引会生成2个段，而段是以区为单位申请存储空间的，一个区默认占用1M存储空间，所以默认情况下一个只存了几条记录的小表也需要2M的存储空间么？以后每次添加一个索引都要多申请2M的存储空间么？这对于存储记录比较少的表简直是天大的浪费。设计 InnoDB 的大叔们都挺节俭的，当然也考虑到了这种情况。这个问题的症结在于到现在为止我们介绍的区都是非常 纯粹 的，也就是一个区被整个分配给某一个段，或者说区中的所有页面都是为了存储同一个段的数据而存在的，即使段的数据填不满区中所有的页面，那余下的页面也不能挪作他用。现在为了考虑以完整的区为单位分配给某个段对于数据量较小的表太浪费存储空间的这种情况，设计 InnoDB 的大叔们提出了一个碎片（fragment）区的概念，也就是在一个碎片区中，并不是所有的页都是为了存储同一个段的数据而存在的，而是碎片区中的页可以用于不同的目的，比如有些页用于段A，有些页用于段B，有些页甚至哪个段都不属于。碎片区直属于表空间，并不属于任何一个段。所以此后为某个段分配存储空间的策略是这样的：

- 在刚开始向表中插入数据的时候，段是从某个碎片区以单个页面为单位来分配存储空间的。
- 当某个段已经占用了32个碎片区页面之后，就会以完整的区为单位来分配存储空间。

所以现在段不能仅定义为是某些区的集合，更精确的应该是某些零散的页面以及一些完整的区的集合。除了索引的叶子节点段和非叶子节点段之外， InnoDB 中还有为存储一些特殊的数据而定义的段，比如回滚段，当然我们现在并不关心别的类型的段，现在只需要知道段是一些零散的页面以及一些完整的区的集合就好了。

9.2.3 区的分类

通过上边一通唠叨，大家知道了表空间的是由若干个区组成的，这些区大体上可以分为4种类型：

- 空闲的区：现在还没有用到这个区中的任何页面。

- 有剩余空间的碎片区：表示碎片区中还有可用的页面。
- 没有剩余空间的碎片区：表示碎片区中的所有页面都被使用，没有空闲页面。
- 附属于某个段的区。每一个索引都可以分为叶子节点段和非叶子节点段，除此之外InnoDB还会另外定义一些特殊作用的段，在这些段中的数据量很大时将使用区来作为基本的分配单位。

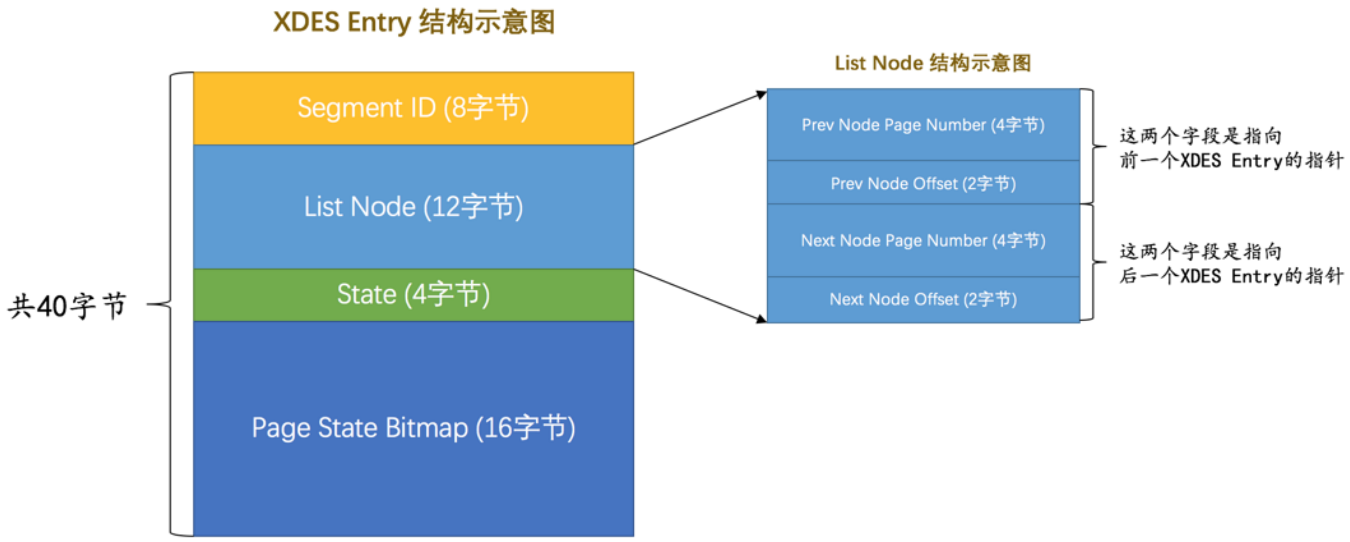
这4种类型的区也可以被称为区的4种状态（ State ），设计 InnoDB 的大叔们为这4种状态的区定义了特定的名词儿：

状态名	含义
FREE	空闲的区
FREE_FRAG	有剩余空间的碎片区
FULL_FRAG	没有剩余空间的碎片区
FSEG	附属于某个段的区

需要再次强调一遍的是，处于 FREE 、 FREE_FRAG 以及 FULL_FRAG 这三种状态的区都是独立的，算是直属于表空间；而处于 FSEG 状态的区是附属于某个段的。

> 小贴士：
> 如果把表空间比作是一个集团军，段就相当于师，区就相当于团。一般的团都是隶属于某个师的，就像是处于`FSEG`的区全都隶属于某个段，而处于`FREE`、`FREE_FRAG`以及`FULL_FRAG`这三种状态的区却直接隶属于表空间，就像独立团直接听命于军部一样。

为了方便管理这些区，设计 InnoDB 的大叔设计了一个称为 XDES Entry 的结构（全称就是Extent Descriptor Entry），每一个区都对应着一个 XDES Entry 结构，这个结构记录了对应的区的一些属性。我们先看图来对这个结构有个大致的了解：

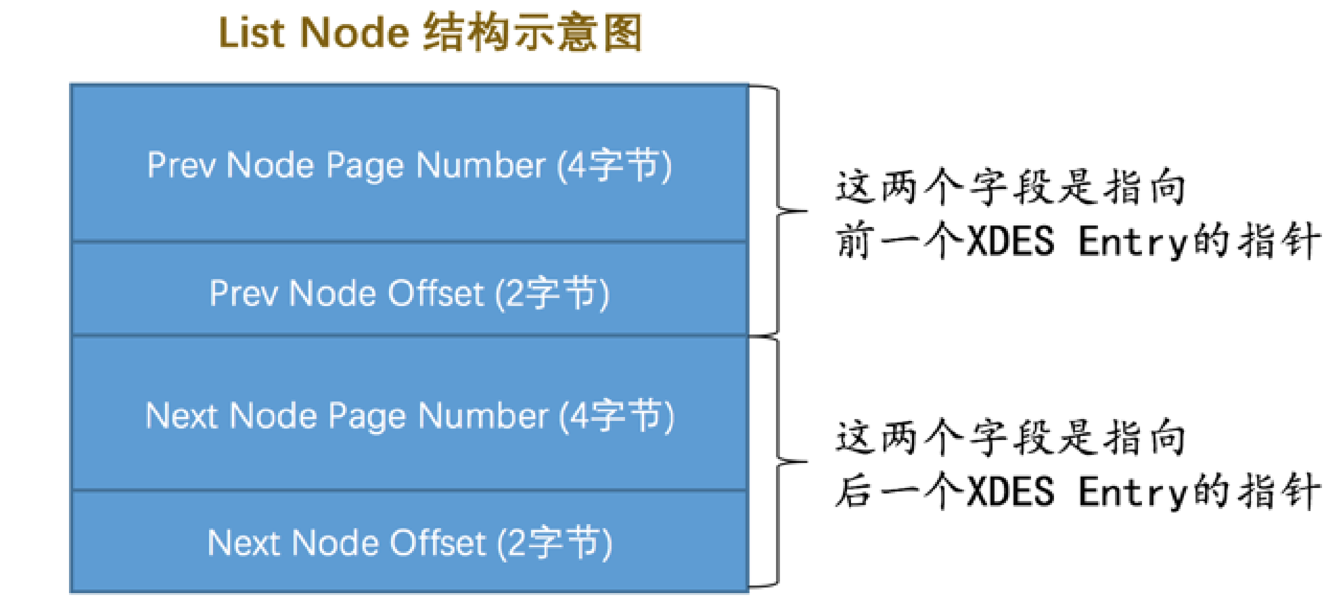


从图中我们可以看出， XDES Entry 是一个40个字节的结构，大致分为4个部分，各个部分的释义如下：

- Segment ID （8字节）

  每一个段都有一个唯一的编号，用ID表示，此处的 Segment ID 字段表示就是该区所在的段。当然前提是该区已经被分配给某个段了，不然的话该字段的值没啥意义。
- List Node （12字节）

  这个部分可以将若干个 XDES Entry 结构串联成一个链表，大家看一下这个 List Node 的结构：



如果我们想定位表空间内的某一个位置的话，只需指定页号以及该位置在指定页号中的页内偏移量即可。所以：

- Pre Node Page Number 和 Pre Node Offset 的组合就是指向前一个 XDES Entry 的指针
- Next Node Page Number 和 Next Node Offset 的组合就是指向后一个 XDES Entry 的指针。

  把一些 XDES Entry 结构连成一个链表有啥用？稍安勿躁，我们稍后唠叨 XDES Entry 结构组成的链表问题。

- State （4字节）

  这个字段表明区的状态。可选的值就是我们前边说过的那4个，分别是： FREE 、 FREE_FRAG 、 FULL_FRAG 和 FSEG 。具体释义就不多唠叨了，前边说的够仔细了。

- Page State Bitmap （16字节）

  这个部分共占用16个字节，也就是128个比特位。我们说一个区默认有64个页，这128个比特位被划分为64个部分，每个部分2个比特位，对应区中的一个页。比如 Page State Bitmap 部分的第1和第2个比特位对应着区中的第1个页面，第3和第4个比特位对应着区中的第2个页面，依此类推， Page State Bitmap 部分的第127和128个比特位对应着区中的第64个页面。这两个比特位的第一个位表示对应的页是否是空闲的，第二个比特位还没有用。

9.2.3.1 XDES Entry链表

到现在为止，我们已经提出了五花八门的概念，什么区、段、碎片区、附属于段的区、 XDES Entry 结构吧啦吧啦的概念，走远了千万别忘了自己为什么出发，我们把事情搞这么麻烦的初心仅仅是想提高向表插入数据的效率又不至于数据量少的表浪费空间。现在我们知道向表中插入数据本质上就是向表中各个索引的叶子节点段、非叶子节点段插入数据，也知道了不同的区有不同的状态，再回到最初的起点，捋一捋向某个段中插入数据的过程：

- 当段中数据较少的时候，首先会查看表空间中是否有状态为 FREE_FRAG 的区，也就是找还有空闲空间的碎片区，如果找到了，那么从该区中取一些零碎的页把数据插进去；否则到表空间下申请一个状态为 FREE 的区，也就是空闲的区，把该区的状态变为 FREE_FRAG ，然后从该新申请的区中取一些零碎的页把数据插进去。之后不同的段使用零碎页的时候都会从该区中取，直到该区中没有空闲空间，然后该区的状态就变成了 FULL_FRAG 。

  现在的问题是你怎么知道表空间里的哪些区是 FREE 的，哪些区的状态是 FREE_FRAG 的，哪些区是 FULL_FRAG 的？要知道表空间的大小是可以不断增大的，当增长到GB级别的时候，区的数量也就上千了，我们总不能每次都遍历这些区对应的 XDES Entry 结构吧？这时候就是 XDES Entry 中的 List Node 部分发挥奇效的时候了，我们可以通过 List Node 中的指针，做这么三件事：

- 把状态为 FREE 的区对应的 XDES Entry 结构通过 List Node 来连接成一个链表，这个链表我们就称之为 FREE 链表。
- 把状态为 FREE_FRAG 的区对应的 XDES Entry 结构通过 List Node 来连接成一个链表，这个链表我们就称之为 FREE_FRAG 链表。
- 把状态为 FULL_FRAG 的区对应的 XDES Entry 结构通过 List Node 来连接成一个链表，这个链表我们就称之为 FULL_FRAG 链表。

  这样每当我们想找一个 FREE_FRAG 状态的区时，就直接把 FREE_FRAG 链表的头节点拿出来，从这个节点中取一些零碎的页来插入数据，当这个节点对应的区用完时，就修改一下这个节点的 State 字段的值，然后从 FREE_FRAG 链表中移到 FULL_FRAG 链表中。同理，如果 FREE_FRAG 链表中一个节点都没有，那么就直接从 FREE 链表中取一个节点移动到 FREE_FRAG 链表的状态，并修改该节点的 STATE 字段值为 FREE_FRAG ，然后从这个节点对应的区中获取零碎的页就好了。

- 当段中数据已经占满了32个零散的页后，就直接申请完整的区来插入数据了。

  还是那个问题，我们怎么知道哪些区属于哪个段的呢？再遍历各个 XDES Entry 结构？遍历是不可能遍历的，这辈子都不可能遍历的，有链表还遍历个毛线啊。所以我们把状态为 FSEG 的区对应的 XDES Entry 结构都加入到一个链表喽？傻呀，不同的段哪能共用一个区呢？你想把索引a的叶子节点段和索引b的叶子节点段都存储到一个区中么？显然我们想要每个段都有它独立的链表，所以可以根据段号（也就是 Segment ID ）来建立链表，有多少个段就建多少个链表？好像也有点问题，因为一个段中可以有好多个区，有的区是完全空闲的，有的区还有一些页面可以用，有的区已经没有空闲页面可以用了，所以我们有必要继续细分，设计 InnoDB 的大叔们为每个段中的区对应的 XDES Entry 结构建立了三个链表：

  - FREE 链表：同一个段中，所有页面都是空闲的区对应的 XDES Entry 结构会被加入到这个链表。注意和直属于表空间的 FREE 链表区别开了，此处的 FREE 链表是附属于某个段的。
  - NOT_FULL 链表：同一个段中，仍有空闲空间的区对应的 XDES Entry 结构会被加入到这个链表。
  - FULL 链表：同一个段中，已经没有空闲空间的区对应的 XDES Entry 结构会被加入到这个链表。

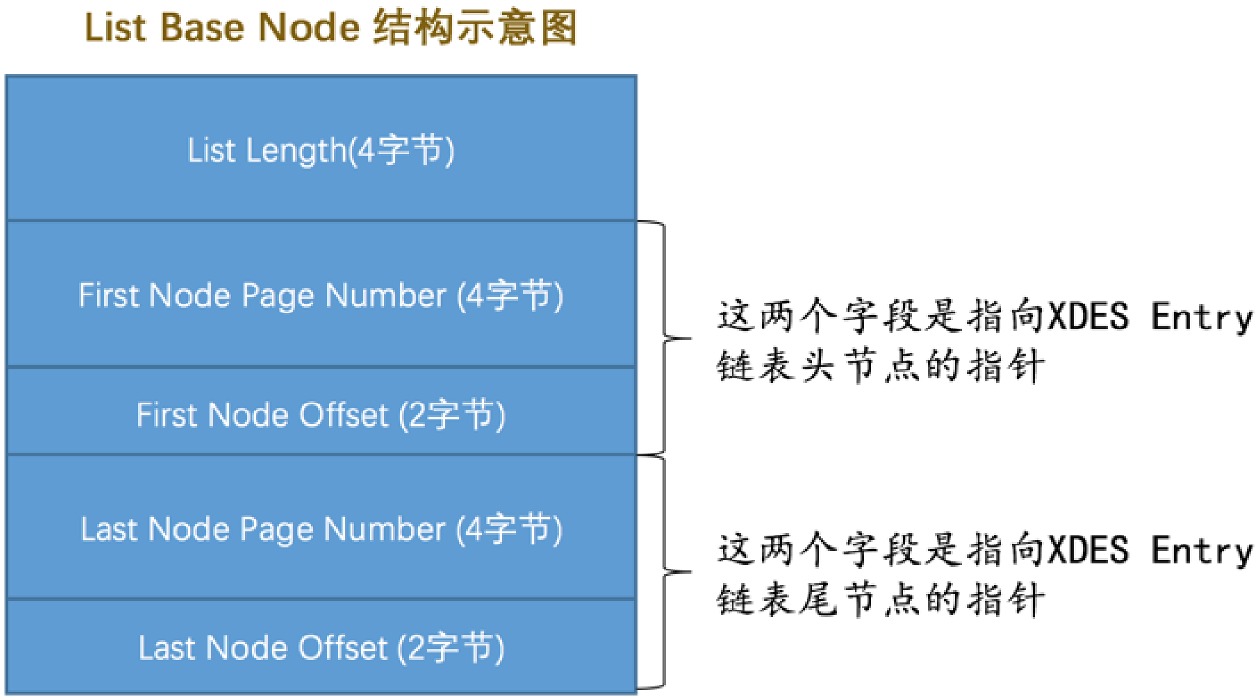
  再次强调一遍，每一个索引都对应两个段，每个段都会维护上述的3个链表，比如下边这个表：

  ```
  CREATE TABLE t (
    c1 INT NOT NULL AUTO_INCREMENT,
    c2 VARCHAR(100),
    c3 VARCHAR(100),
    PRIMARY KEY (c1),
    KEY idx_c2 (c2)
  )ENGINE=InnoDB;
  ```

  这个表 t 共有两个索引，一个聚簇索引，一个二级索引 idx_c2 ，所以这个表共有4个段，每个段都会维护上述3个链表，总共是12个链表，加上我们上边说过的直属于表空间的3个链表，整个独立表空间共需要维护15个链表。所以段在数据量比较大时插入数据的话，会先获取 NOT_FULL 链表的头节点，直接把数据插入这个头节点对应的区中即可，如果该区的空间已经被用完，就把该节点移到 FULL 链表中。

9.2.3.2 链表基节点

上边光是介绍了一堆链表，可我们怎么找到这些链表呢，或者说怎么找到某个链表的头节点或者尾节点在表空间中的位置呢？设计 InnoDB 的大叔当然考虑了这个问题，他们设计了一个叫 List Base Node 的结构，翻译成中文就是链表的基节点。这个结构中包含了链表的头节点和尾节点的指针以及这个链表中包含了多少节点的信息，我们画图看一下这个结构的示意图：



我们上边介绍的每个链表都对应这么一个 `List Base Node` 结构，其中：

- `List Length` 表明该链表一共有多少节点，
- `First Node Page Number` 和 `First Node Offset` 表明该链表的头节点在表空间中的位置。
- `Last Node Page Number` 和 `Last Node Offset` 表明该链表的尾节点在表空间中的位置。

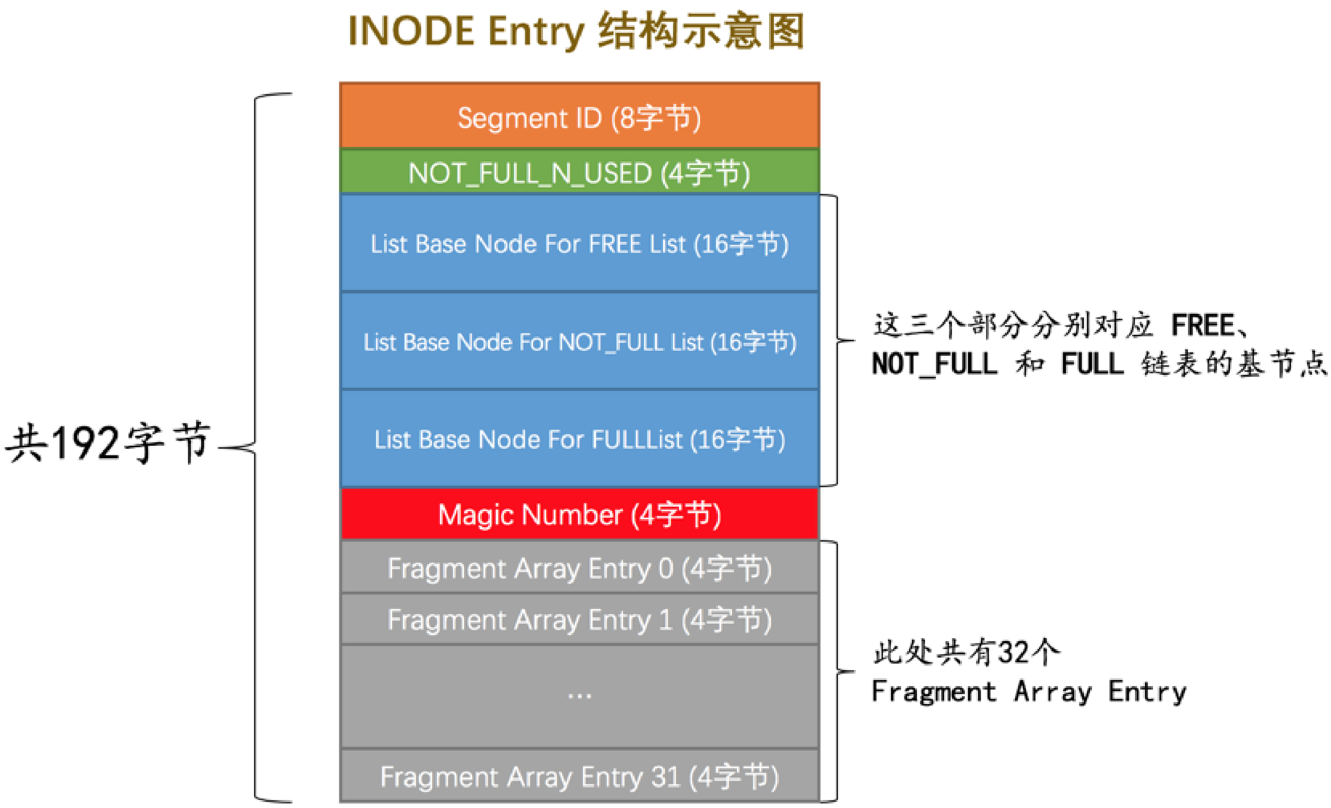
一般我们把某个链表对应的 `List Base Node` 结构放置在表空间中固定的位置，这样想找定位某个链表就变得so easy啦。

9.2.3.3 链表小结

综上所述，表空间是由若干个区组成的，每个区都对应一个 `XDES Entry` 的结构，直属于表空间的区对应的 `XDES Entry` 结构可以分成 `FREE` 、 `FREE_FRAG` 和 `FULL_FRAG` 这3个链表；每个段可以附属若干个区，每个段中的区对应的 `XDES Entry` 结构可以分成 `FREE` 、 `NOT_FULL` 和 `FULL` 这3个链表。每个链表都对应一个 `List Base Node` 的结构，这个结构里记录了链表的头、尾节点的位置以及该链表中包含的节点数。正是因为这些链表的存在，管理这些区才变成了一件so easy的事情。

9.2.4 段的结构

我们前边说过，段其实不对应表空间中某一个连续的物理区域，而是一个逻辑上的概念，由若干个零散的页面以及一些完整的区组成。像每个区都有对应的 `XDES Entry` 来记录这个区中的属性一样，设计 `InnoDB` 的大叔为每个段都定义了一个 `INODE Entry` 结构来记录一下段中的属性。大家看一下示意图：



它的各个部分释义如下：

- Segment ID

  就是指这个 INODE Entry 结构对应的段的编号（ID）。
- NOT_FULL_N_USED

  这个字段指的是在 NOT_FULL 链表中已经使用了多少个页面。下次从 NOT_FULL 链表分配空闲页面时可以直接根据这个字段的值定位到。而不用从链表中的第一个页面开始遍历着寻找空闲页面。
- 3个 List Base Node

  分别为段的 FREE 链表、 NOT_FULL 链表、 FULL 链表定义了 List Base Node ，这样我们想查找某个段的某个链表的头节点和尾节点的时候，就可以直接到这个部分找到对应链表的 List Base Node 。so easy!
- Magic Number ：

  这个值是用来标记这个 INODE Entry 是否已经被初始化了（初始化的意思就是把各个字段的值都填进去了）。如果这个数字是值的 97937874 ，表明该 INODE Entry 已经初始化，否则没有被初始化。（不用纠结这个值有啥特殊含义，人家规定的）。
- Fragment Array Entry

  我们前边强调过无数次段是一些零散页面和一些完整的区的集合，每个 Fragment Array Entry 结构都对应着一个零散的页面，这个结构一共4个字节，表示一个零散页面的页号。

结合着这个 INODE Entry 结构，大家可能对段是一些零散页面和一些完整的区的集合的理解再次深刻一些。

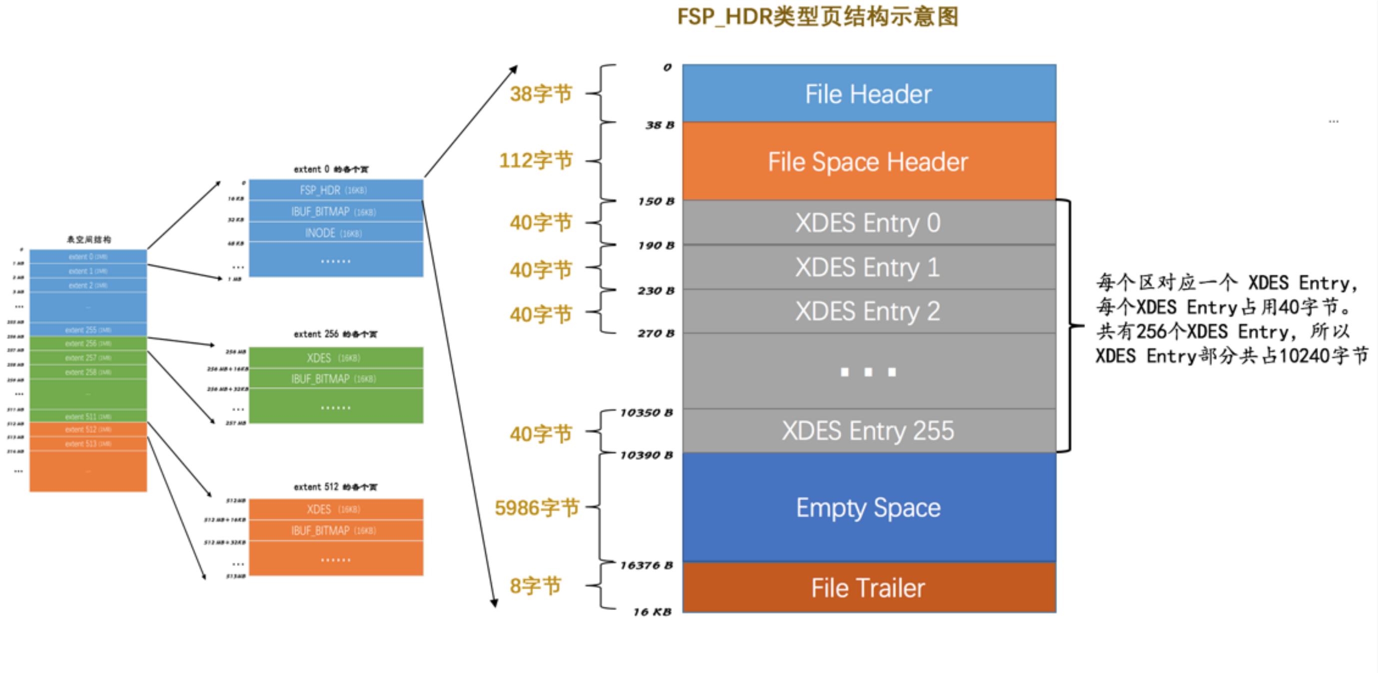
9.2.5 各类型页面详细情况

到现在为止我们已经大概清楚了表空间、段、区、XDES Entry、INODE Entry、各种以 XDES Enty 为节点的链表的基本概念了，可是总有一种飞在天上不踏实的感觉，每个区对应的 XDES Entry 结构到底存储在表空间的什么地方？直属于表空间的 FREE 、 FREE_FRAG 、 FULL_FRAG 链表的基节点到底存储在表空间的什么地方？每个段对

应的 INODE Entry 结构到底存在表空间的什么地方？我们前边介绍了每256个连续的区算是一个组，想解决刚才提出来的这些个疑问还得从每个组开头的一些类型相同的页面说起，接下来我们一个页面一个页面的分析，真相马上就要浮出水面了。

9.2.5.1 FSP_HDR 类型

首先看第一个组的第一个页面，当然也是表空间的第一个页面，页号为 0 。这个页面的类型是 FSP_HDR ，它存储了表空间的一些整体属性以及第一个组内256个区的对应的 XDES Entry 结构，直接看这个类型的页面的示意图：



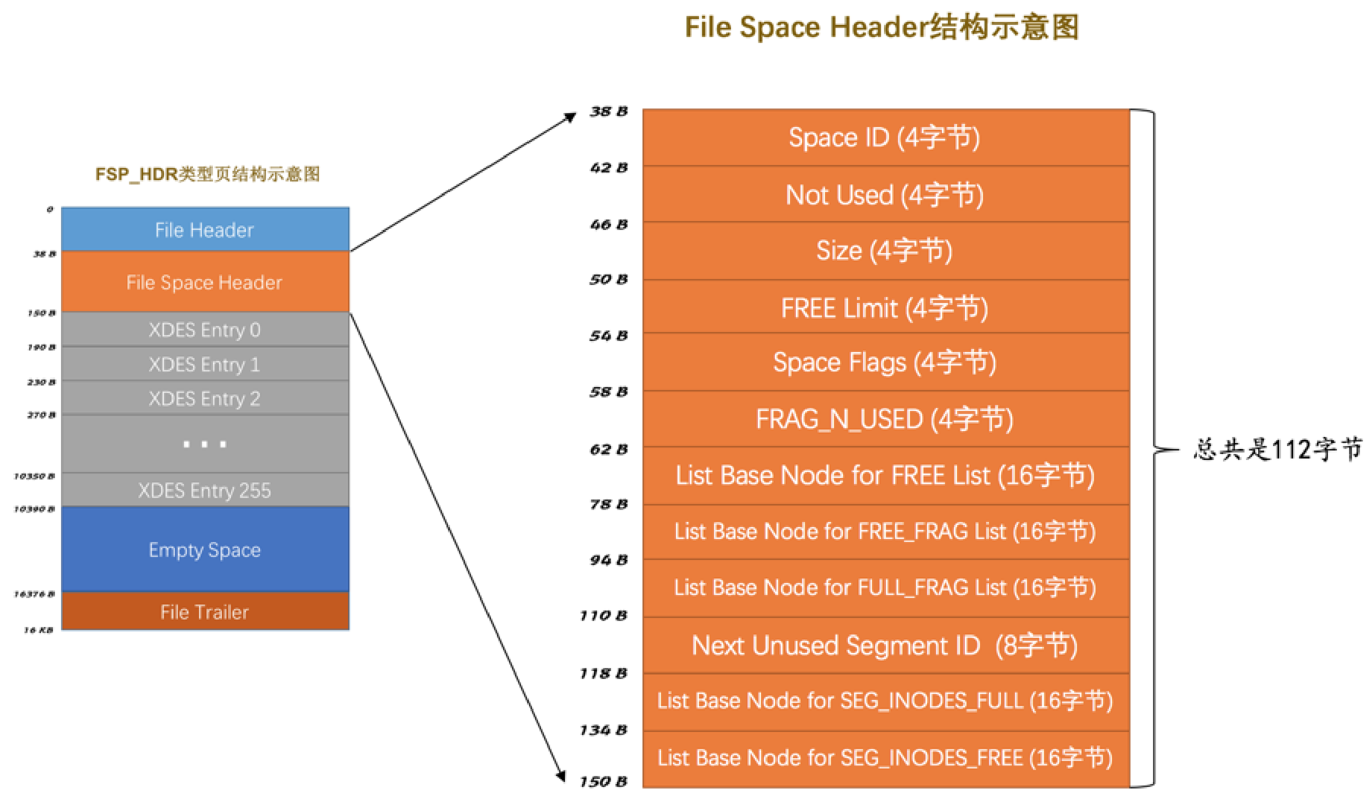
从图中可以看出，一个完整的 FSP_HDR 类型的页面大致由5个部分组成，各个部分的具体释义如下表：

名称	中文名	占用空间大小	简单描述
File Header	文件头部	38 字节	页的一些通用信息
File Space Header	表空间头部	112 字节	表空间的一些整体属性信息
XDES Entry	区描述信息	10240 字节	存储本组256个区对应的属性信息
Empty Space	尚未使用空间	5986 字节	用于页结构的填充，没啥实际意义
File Trailer	文件尾部	8 字节	校验页是否完整

File Header 和 File Trailer 就不再强调了，另外的几个部分中， Empty Space 是尚未使用的空间，我们不用管它，重点来看看 File Space Header 和 XDES Entry 这两个部分。

File Space Header部分

从名字就可以看出来，这个部分是用来存储表空间的一些整体属性的，废话少说，看图：



哇唔，字段有点儿多哦，不急一个一个慢慢看。下面是各个属性的简单描述：

名称	占用空间大小	描述
Space ID	4 字节	表空间的ID
Not Used	4 字节	这4个字节未被使用，可以忽略
Size	4 字节	当前表空间占有的页面数
FREE Limit	4 字节	尚未被初始化的最小页号，大于或等于这个页号的区对应的XDES Entry结构都没有被加入FREE链表
Space Flags	4 字节	表空间的一些占用存储空间比较小的属性
FRAG_N_USED	4 字节	FREE_FRAG链表中已使用的页面数量
List Base Node for FREE List	16 字节	FREE链表的基节点
List Base Node for FREE_FRAG List	16 字节	FREE_FREG链表的基节点
List Base Node for FULL_FRAG List	16 字节	FULL_FREG链表的基节点
Next Unused Segment ID	8 字节	当前表空间中下一个未使用的 Segment ID
List Base Node for SEG_INODES_FULL List	16 字节	SEG_INODES_FULL链表的基节点
List Base Node for SEG_INODES_FREE List	16 字节	SEG_INODES_FREE链表的基节点

这里头的 Space ID 、 Not Used 、 Size 这三个字段大家肯定一看就懂，其他的字段我们再详细瞅瞅，为了大家的阅读体验，我就不严格按照实际的字段顺序来解释各个字段了哈。

- List Base Node for FREE List 、 List Base Node for FREE_FRAG List 、 List Base Node for FULL_FRAG List 。

这三个大家看着太亲切了，分别是直属于表空间的 FREE 链表的基节点、 FREE_FRAG 链表的基节点、 FULL_FRAG 链表的基节点，这三个链表的基节点在表空间的位置是固定的，就是在表空间的第一个页面（也就是 FSP_HDR 类型的页面）的 File Space Header 部分。所以之后定位这几个链表就so easy啦。

- FRAG_N_USED

  这个字段表明在 FREE_FRAG 链表中已经使用的页面数量，方便之后在链表中查找空闲的页面。

- FREE Limit

  我们知道表空间都对应着具体的磁盘文件，一开始我们创建表空间的时候对应的磁盘文件中都没有数据，所以我们需要对表空间完成一个初始化操作，包括为表空间中的区建立 XDES Entry 结构，为各个段建立 INODE Entry 结构，建立各种链表吧啦吧啦的各种操作。我们可以一开始就为表空间申请一个特别大的空间，但是实际上有绝大部分的区是空闲的，我们可以选择把所有的这些空闲区对应的 XDES Entry 结构加入 FREE 链表，也可以选择只把一部分的空闲区加入 FREE 链表，等啥时候空闲链表中的 XDES Entry 结构对应的区不够使了，再把之前没有加入 FREE 链表的空闲区对应的 XDES Entry 结构加入 FREE 链表，中心思想就是啥时候用到啥时候初始化，设计 InnoDB 的大叔采用的就是后者，他们为表空间定义了 FREE Limit 这个字段，在该字段表示的页号之前的区都被初始化了，之后的区尚未被初始化。

- Next Unused Segment ID

  表中每个索引都对应2个段，每个段都有一个唯一的ID，那当我们为某个表新创建一个索引的时候，就意味着要创建两个新的段。那怎么为这个新创建的段找一个唯一的ID呢？去遍历现在表空间中所有的段么？我们说过，遍历是不可能遍历的，这辈子都不可能遍历，所以设计 InnoDB 的大叔们提出了这个名叫 Next Unused Segment ID 的字段，该字段表明当前表空间中最大的段ID的下一个ID，这样在创建新段的时候赋予新段一个唯一的ID值就so easy啦，直接使用这个字段的值就好了。

- Space Flags

  表空间对于一些布尔类型的属性，或者只需要寥寥几个比特位搞定的属性都放在了这个 Space Flags 中存储，虽然它只有4个字节，32个比特位大小，却存储了好多表空间的属性，详细情况如下表：

  |标志名称|占用的空间（单位：bit）|描述| |:--:|:--:|:--:| | POST_ANTELOPE |1|表示文件格式是否大于 ANTELOPE | | ZIP_SSIZE |4|表示压缩页面的大小| | ATOMIC_BLOBS |1|表示是否自动把值非常长的字段放到BLOB页里| | PAGE_SSIZE |4|页面大小| | DATA_DIR |1|表示表空间是否是从默认的数据目录中获取的| | SHARED |1|是否为共享表空间| | TEMPORARY |1|是否为临时表空间| | ENCRYPTION |1|表空间是否加密| | UNUSED |18|没有使用到的比特位|

  > 小贴士：
  > 不同MySQL版本里 SPACE_FLAGS 代表的属性可能有些差异，我们这里列举的是5.7.21版本的。不过大家现在不必深究它们的意思，因为我们一旦把这些概念展开，就需要非常大的篇幅，主要怕大家受不了。我们还是先挑重要的看，把主要的表空间结构了解完，这些 SPACE_FLAGS 里的属性的细节就暂时不深究了。

- List Base Node for SEG_INODES_FULL List 和 List Base Node for SEG_INODES_FREE List

  每个段对应的 INODE Entry 结构会集中存放到一个类型位 INODE 的页中，如果表空间中的段特别多，则会有多个 INODE Entry 结构，可能一个页放不下，这些 INODE 类型的页会组成两种列表：

  - SEG_INODES_FULL 链表，该链表中的 INODE 类型的页面都已经被 INODE Entry 结构填充满了，没空闲空间存放额外的 INODE Entry 了。
  - SEG_INODES_FREE 链表，该链表中的 INODE 类型的页面都已经仍有空闲空间来存放 INODE Entry 结构。

  由于我们现在还没有详细唠叨 INODE 类型页，所以等会说过 INODE 类型的页之后再回过头来看着两个链表。

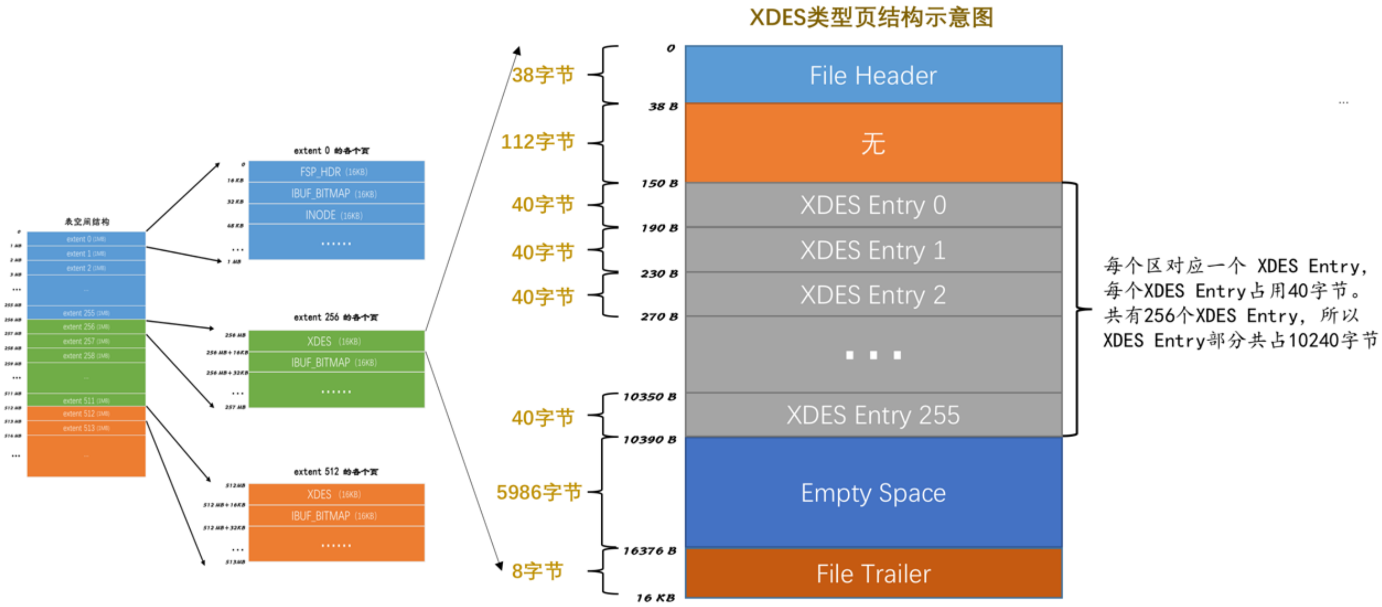
XDES Entry部分

紧接着 File Space Header 部分的就是 XDES Entry 部分了，我们嘴上唠叨过无数次，却从没见过真身的 XDES Entry 就是在表空间的第一个页面中保存的。我们知道一个 XDES Entry 结构的大小是40字节，但是一个页面的大小有限，只能存放有限个 XDES Entry 结构，所以我们才把256个区划分成一组，在每组的第一个页面中存放256个 XDES Entry 结构。大家回看那个 FSP_HDR 类型页面的示意图， XDES Entry 0 就对应着 extent 0 ， XDES Entry 1 就对应着 extent 1 ... 依此类推， XDES Entry255 就对应着 extent 255 。

因为每个区对应的 XDES Entry 结构的地址是固定的，所以我们访问这些结构就so easy啦，至于该结构的详细使用情况我们已经唠叨的够明白了，在这就不赘述了。

9.2.5.2 XDES 类型

我们说过，每一个 XDES Entry 结构对应表空间的一个区，虽然一个 XDES Entry 结构只占用40字节，但你抵不住表空间的区的数量也多啊。在区的数量非常多时，一个单独的页可能就不够存放足够多的 XDES Entry 结构，所以我们把表空间的区分为了若干个组，每组开头的一个页面记录着本组内所有的区对应的 XDES Entry 结构。由于第一个组的第一个页面有些特殊，因为它也是整个表空间的第一个页面，所以除了记录本组中的所有区对应的 XDES Entry 结构以外，还记录着表空间的一些整体属性，这个页面的类型就是我们刚刚说完的 FSP_HDR 类型，整个表空间里只有一个这个类型的页面。除去第一个分组以外，之后的每个分组的第一个页面只需要记录本组内所有的区对应的 XDES Entry 结构即可，不需要再记录表空间的属性了，为了和 FSP_HDR 类型做区别，我们把之后每个分组的第一个页面的类型定义为 XDES ，它的结构和 FSP_HDR 类型是非常相似的：



与 FSP_HDR 类型的页面对比，除了少了 File Space Header 部分之外，也就是除了少了记录表空间整体属性的部分之外，其余的部分是一样一样的。由于我们上边唠叨的已经够仔细了，对于 XDES 类型的页面也就不重复唠叨了哈。

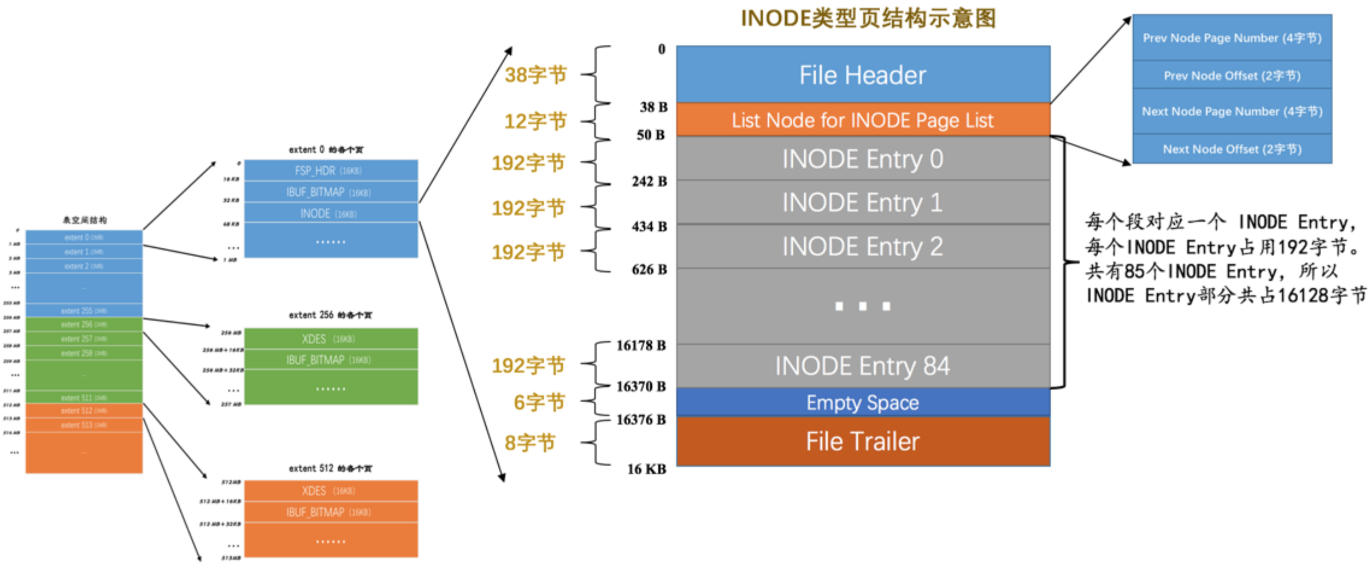
9.2.5.3 IBUF_BITMAP 类型

对比前边介绍表空间的图，每个分组的第二个页面的类型都是 IBUF_BITMAP ，这种类型的页里边记录了一些有关 Change Buffer 的东东，由于这个 Change Buffer 里又包含了贼多的概念，考虑到大家在一章中接受这么多新概念有点呼吸不适，怕大家心脏病犯了所以就把 Change Buffer 的相关知识放到后边的章节中，大家稍安勿躁哈。

9.2.5.4 INODE 类型

再次对比前边介绍表空间的图，第一个分组的第三个页面的类型是 INODE 。我们前边说过设计 InnoDB 的大叔为每个索引定义了两个段，而且为某些特殊功能定义了些特殊的段。为了方便管理，他们又为每个段设计了一个 INODE Entry 结构，这个结构中记录了关于这个段的相关属性。而我们这会儿要介绍的这个 INODE 类型的页就是

为了存储 INODE Entry 结构而存在的。好了，废话少说，直接看图：



从图中可以看出，一个 INODE 类型的页面是由这几部分构成的：

名称	中文名	占用空间大小	简单描述
File Header	文件头部	38 字节	页的一些通用信息
List Node for INODE Page List	通用链表节点	12 字节	存储上一个INODE页面和下一个INODE页面的指针
INODE Entry	段描述信息	16128 字节	
Empty Space	尚未使用空间	6 字节	用于页结构的填充，没啥实际意义
File Trailer	文件尾部	8 字节	校验页是否完整

除了 File Header 、 Empty Space 、 File Trailer 这几个老朋友外，我们重点关注 List Node for INODE Page List 和 INODE Entry 这两个部分。

首先看 INODE Entry 部分，我们前边已经详细介绍过这个结构的组成了，主要包括对应的段内零散页面的地址以及附属于该段的 FREE 、 NOT_FULL 和 FULL 链表的基节点。每个 INODE Entry 结构占用192字节，一个页面里可以存储 85 个这样的结构。

重点看一下 List Node for INODE Page List 这个玩意儿，因为一个表空间中可能存在超过85个段，所以可能一个 INODE 类型的页面不足以存储所有的段对应的 INODE Entry 结构，所以就需要额外的 INODE 类型的页面来存储这些结构。还是为了方便管理这些 INODE 类型的页面，设计 InnoDB 的大叔们将这些 INODE 类型的页面串联成两个不同的链表：

- SEG_INODES_FULL 链表：该链表中的 INODE 类型的页面中已经没有空闲空间来存储额外的 INODE Entry 结构了。
- SEG_INODES_FREE 链表：该链表中的 INODE 类型的页面中还有空闲空间来存储额外的 INODE Entry 结构了。

想必大家已经认出这两个链表了，我们前边提到过这两个链表的基节点就存储在 File Space Header 里边，也就是说这两个链表的基节点的位置是固定的，所以我们可以很轻松的访问到这两个链表。以后每当我们新创建一个段（创建索引时就会创建段）时，都会创建一个 INODE Entry 结构与之对应，存储 INODE Entry 的大致过程就是这样的：

- 先看看 SEG_INODES_FREE 链表是否为空，如果不为空，直接从该链表中获取一个节点，也就相当于获取到一个仍有空闲空间的 INODE 类型的页面，然后把该 INODE Entry 结构防到该页面中。当该页面中无剩余空间时，就把该页放到 SEG_INODES_FULL 链表中。
- 如果 SEG_INODES_FREE 链表为空，则需要从表空间的 FREE_FRAG 链表中申请一个页面，修改该页面的类型为 INODE ，把该页面放到 SEG_INODES_FREE 链表中，与此同时把该 INODE Entry 结构放入该页面。

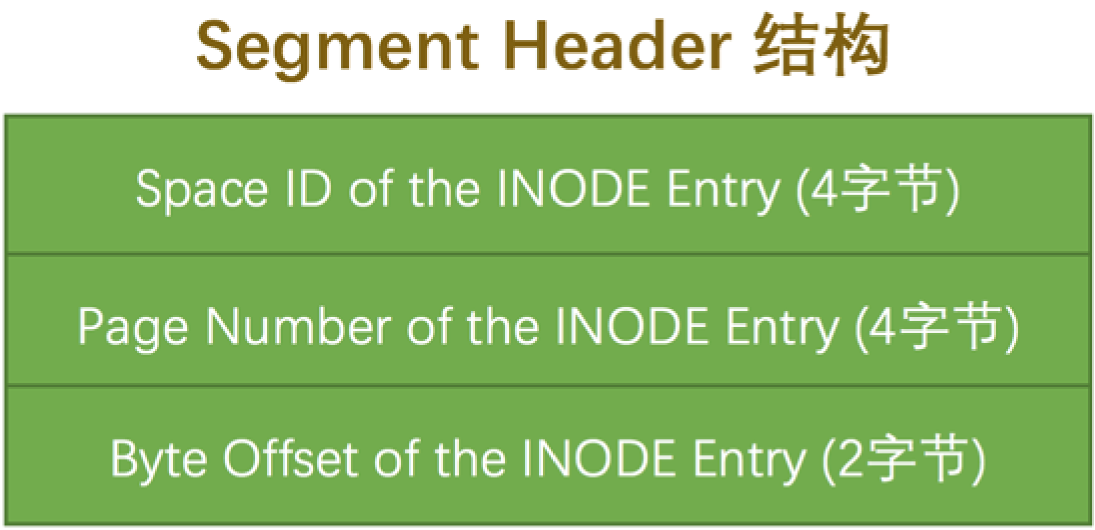
9.2.6 Segment Header 结构的运用

我们知道一个索引会产生两个段，分别是叶子节点段和非叶子节点段，而每个段都会对应一个 INODE Entry 结构，那我们怎么知道某个段对应哪个 INODE Entry 结构呢？所以得找个地方记下来这个对应关系。希望你还记得我们在唠叨数据页，也就是 INDEX 类型的页时有一个 Page Header 部分，当然我不能指望你记住，所以把 Page Header 部分再抄一遍给你看：

Page Header部分（为突出重点，省略了好多属性）

名称	占用空间大小	描述
...	...	...
PAGE_BTR_SEG_LEAF	10 字节	B+树叶子段的头部信息，仅在B+树的根页定义
PAGE_BTR_SEG_TOP	10 字节	B+树非叶子段的头部信息，仅在B+树的根页定义

其中的 PAGE_BTR_SEG_LEAF 和 PAGE_BTR_SEG_TOP 都占用10个字节，它们其实对应一个叫 Segment Header 的结构，该结构图示如下：



各个部分的具体释义如下：

名称	占用字节数	描述
Space ID of the INODE Entry	4	INODE Entry结构所在的表空间ID
Page Number of the INODE Entry	4	INODE Entry结构所在的页面页号
Byte Offset of the INODE Ent	2	INODE Entry结构在该页面中的偏移量

这样子就很清晰了， PAGE_BTR_SEG_LEAF 记录着叶子节点段对应的 INODE Entry 结构的地址是哪个表空间的哪个页面的哪个偏移量， PAGE_BTR_SEG_TOP 记录着非叶子节点段对应的 INODE Entry 结构的地址是哪个表空间的哪个页面的哪个偏移量。这样子索引和其对应的段的关系就建立起来了。不过需要注意的一点是，因为一个索引只对应两个段，所以只需要在索引的根页面中记录这两个结构即可。

9.2.7 真实表空间对应的文件大小

等会儿等会儿，上边的这些概念已经压的快喘不过气了。不过独立表空间有那么大么？我到数据目录里看了，一个新建的表对应的 .ibd 文件只占用了96K，才6个页面大小，上边的内容该不是扯犊子吧？

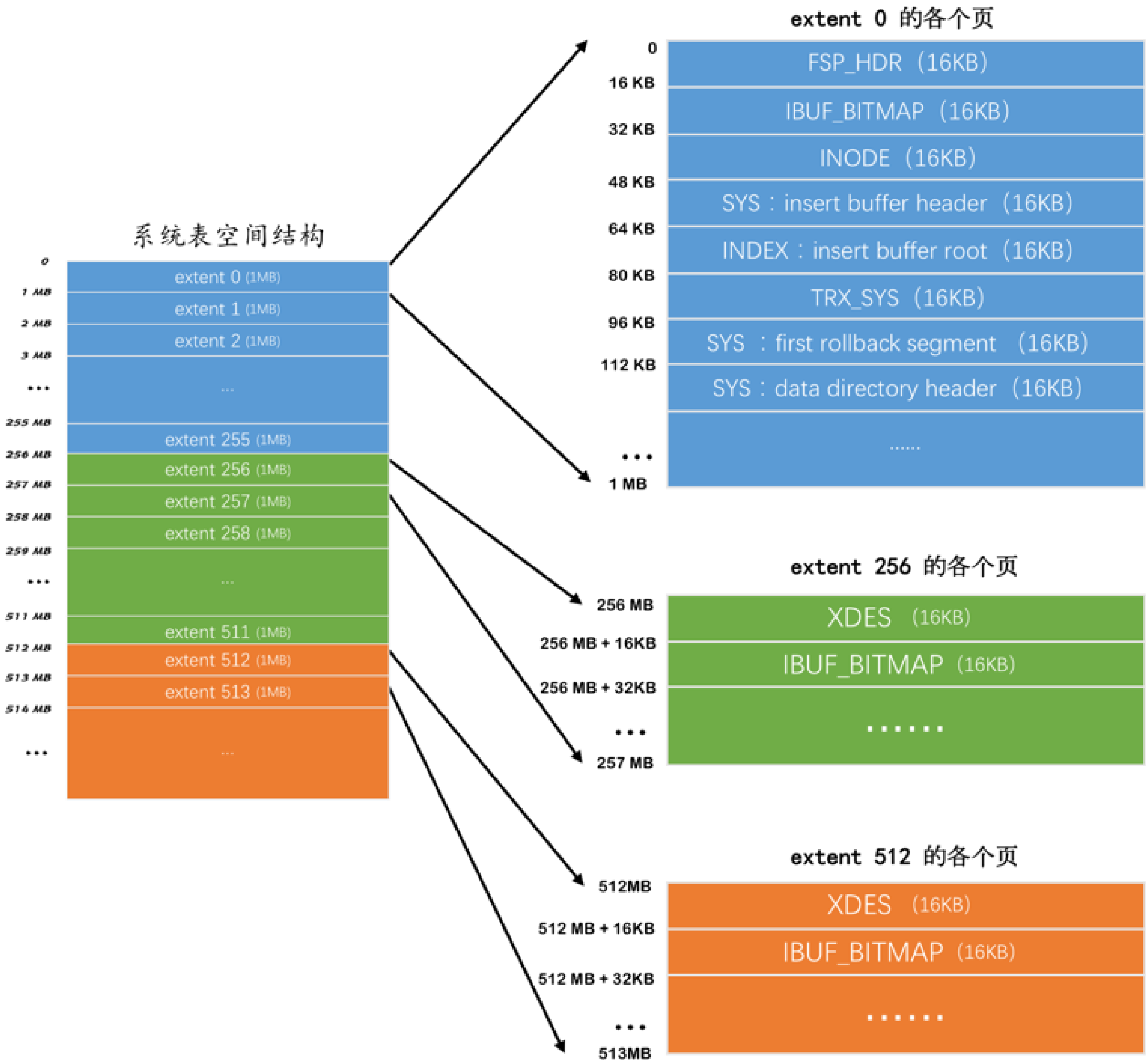
哈，一开始表空间占用的空间自然是很小，因为表里边都没有数据嘛！不过别忘了这些 .ibd 文件是自扩展的，随着表中数据的增多，表空间对应的文件也逐渐增大。

9.3 系统表空间

了解完了独立表空间的基本结构，系统表空间的结构也就好理解多了，系统表空间的结构和独立表空间基本类似，只不过由于整个MySQL进程只有一个系统表空间，在系统表空间中会额外记录一些有关整个系统信息的页面，所以会比独立表空间多出一些记录这些信息的页面。因为这个系统表空间最牛逼，相当于是表空间之首，所以它的 表空间 ID （Space ID）是 0 。

9.3.1 系统表空间的整体结构

系统表空间与独立表空间的一个非常明显的不同之处就是在表空间开头有许多记录整个系统属性的页面，如图：



可以看到，系统表空间和独立表空间的前三个页面（页号分别为 0 、 1 、 2 ，类型分别是 FSP_HDR 、 IBUF_BITMAP 、 INODE ）的类型是一致的，只是页号为 3 ～ 7 的页面是系统表空间特有的，我们来看一下这些多出来的页面都是干啥使的：

页号	页面类型	英文描述	描述
3	SYS	Insert Buffer Header	存储Insert Buffer的头部信息
4	INDEX	Insert Buffer Root	存储Insert Buffer的根页面
5	TRX_SYS	Transction System	事务系统的相关信息
6	SYS	First Rollback Segment	第一个回滚段的页面
7	SYS	Data Dictionary Header	数据字典头部信息

除了这几个记录系统属性的页面之外，系统表空间的 `extent 1` 和 `extent 2` 这两个区，也就是页号从 `64` ~ `191` 这128个页面被称为 `Doublewrite buffer` ，也就是双写缓冲区。不过上述的大部分知识都涉及到了事务和多版本控制的问题，这些问题我们会放在后边的章节集中唠叨，现在讲述太影响用户体验，所以现在我们只唠叨一下有关InnoDB数据字典的知识，其余的概念在后边再看。

9.3.1.1 InnoDB数据字典

我们平时使用 `INSERT` 语句向表中插入的那些记录称之为用户数据，MySQL只是作为一个软件来为我们来保管这些数据，提供方便的增删改查接口而已。但是每当我们向一个表中插入一条记录的时候，MySQL先要校验一下插入语句对应的表存不存在，插入的列和表中的列是否符合，如果语法没有问题的话，还需要知道该表的聚簇索引和所有二级索引对应的根页面是哪个表空间的哪个页面，然后把记录插入对应索引的 `B+` 树中。所以说，MySQL除了保存着我们插入的用户数据之外，还需要保存许多额外的信息，比方说：

- 某个表属于哪个表空间，表里边有多少列
- 表对应的每一个列的类型是什么
- 该表有多少索引，每个索引对应哪几个字段，该索引对应的根页面在哪个表空间的哪个页面
- 该表有哪些外键，外键对应哪个表的哪些列
- 某个表空间对应文件系统上文件路径是什么
- balabala ... 还有好多，不一一列举了

上述这些数据并不是我们使用 `INSERT` 语句插入的用户数据，实际上是为了更好的管理我们这些用户数据而不得已引入的一些额外数据，这些数据也称为 `元数据` 。InnoDB存储引擎特意定义了一些列的内部系统表（internal system table）来记录这些这些 `元数据` ：

表名	描述
SYS_TABLES	整个InnoDB存储引擎中所有的表的信息
SYS_COLUMNS	整个InnoDB存储引擎中所有的列的信息
SYS_INDEXES	整个InnoDB存储引擎中所有的索引的信息
SYS_FIELDS	整个InnoDB存储引擎中所有的索引对应的列的信息
SYS_FOREIGN	整个InnoDB存储引擎中所有的外键的信息
SYS_FOREIGN_COLS	整个InnoDB存储引擎中所有的外键对应列的信息
SYS_TABLESPACES	整个InnoDB存储引擎中所有的表空间信息
SYS_DATAFILES	整个InnoDB存储引擎中所有的表空间对应文件系统的文件路径信息
SYS_VIRTUAL	整个InnoDB存储引擎中所有的虚拟生成列的信息

这些系统表也被称为 `数据字典` ，它们都是以 `B+` 树的形式保存在系统表空间的某些页面中，其中 `SYS_TABLES` 、 `SYS_COLUMNS` 、 `SYS_INDEXES` 、 `SYS_FIELDS` 这四个表尤其重要，称之为基本系统表（basic system tables），我们先看看这4个表的结构：

SYS_TABLES表

SYS_TABLES表的列

列名	描述
NAME	表的名称
ID	InnoDB存储引擎中每个表都有一个唯一的ID
N_COLS	该表拥有列的个数
TYPE	表的类型，记录了一些文件格式、行格式、压缩等信息
MIX_ID	已过时，忽略
MIX_LEN	表的一些额外的属性
CLUSTER_ID	未使用，忽略
SPACE	该表所属表空间的ID

这个 SYS_TABLES 表有两个索引:

- 以 NAME 列为主键的聚簇索引
- 以 ID 列建立的二级索引

SYS_COLUMNS表

SYS_COLUMNS表的列

列名	描述
TABLE_ID	该列所属表对应的ID
POS	该列在表中是第几列
NAME	该列的名称
MTYPE	main data type，主数据类型，就是那堆INT、CHAR、VARCHAR、FLOAT、DOUBLE之类的东东
PRTYPE	precise type，精确数据类型，就是修饰主数据类型的那堆东东，比如是否允许NULL值，是否允许负数啥的
LEN	该列最多占用存储空间的字节数
PREC	该列的精度，不过这列貌似都没有使用，默认值都是0

这个 SYS_COLUMNS 表只有一个聚集索引:

- 以 (TABLE_ID, POS) 列为主键的聚簇索引

SYS_INDEXES表

SYS_INDEXES表的列

列名	描述
TABLE_ID	该索引所属表对应的ID
ID	InnoDB存储引擎中每个索引都有一个唯一的ID
NAME	该索引的名称
N_FIELDS	该索引包含列的个数
TYPE	该索引的类型，比如聚簇索引、唯一索引、更改缓冲区的索引、全文索引、普通的二级索引等等各种类型
SPACE	该索引根页面所在的表空间ID
PAGE_NO	该索引根页面所在的页面号

列名	描述
MERGE_THRESHOLD	如果页面中的记录被删除到某个比例，就把该页面和相邻页面合并，这个值就是这个比例

这个 SYS_INEXES 表只有一个聚集索引：

- 以 (TABLE_ID, ID) 列为主键的聚簇索引

SYS_FIELDS表

SYS_FIELDS表的列

列名	描述
INDEX_ID	该索引列所属的索引的ID
POS	该索引列在某个索引中是第几列
COL_NAME	该索引列的名称

这个 SYS_INEXES 表只有一个聚集索引：

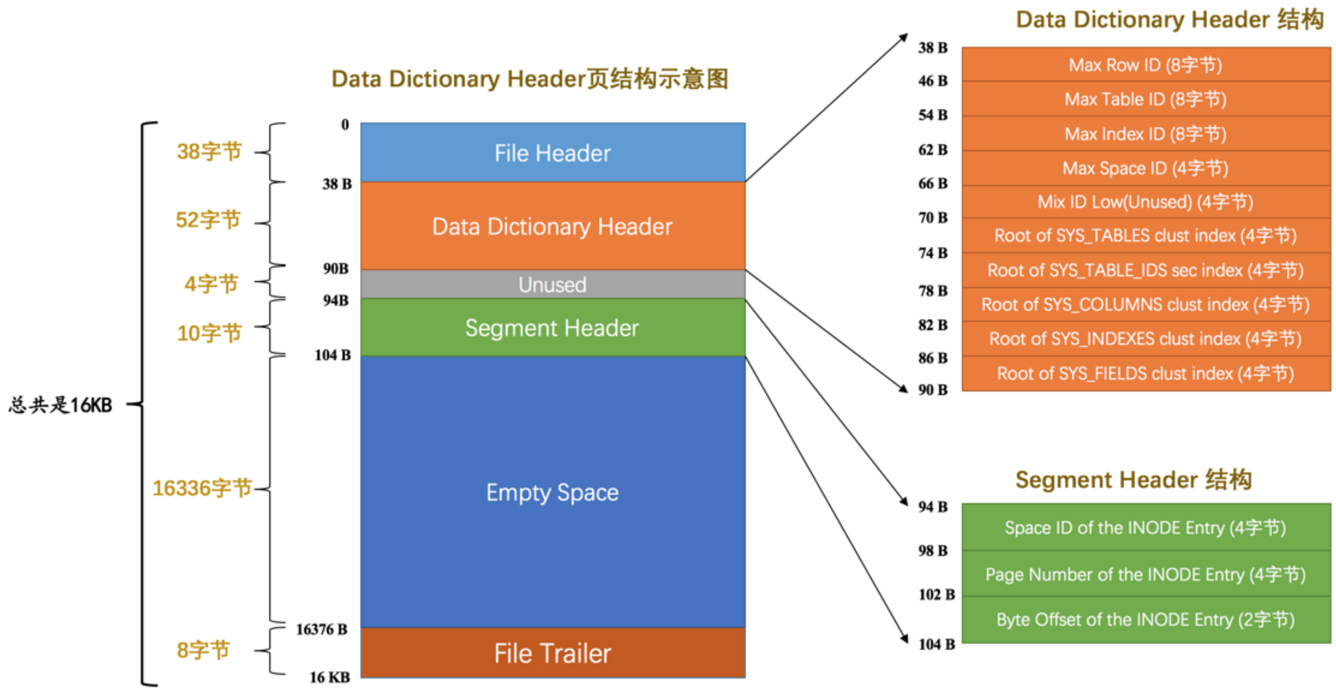
- 以 (INDEX_ID, POS) 列为主键的聚簇索引

Data Dictionary Header页面

只要有了上述4个基本系统表，也就意味着可以获取其他系统表以及用户定义的表的所有元数据。比方说我们想看看 SYS_TABLESPACES 这个系统表里存储了哪些表空间以及表空间对应的属性，那就可以：

- 到 SYS_TABLES 表中根据表名定位到具体的记录，就可以获取到 SYS_TABLESPACES 表的 TABLE_ID
- 使用这个 TABLE_ID 到 SYS_COLUMNS 表中就可以获取到属于该表的所有列的信息。
- 使用这个 TABLE_ID 还可以到 SYS_INDEXES 表中获取所有的索引的信息，索引的信息中包括对应的 INDEX_ID ，还记录着该索引对应的 B+ 数根页面是哪个表空间的哪个页面。
- 使用 INDEX_ID 就可以到 SYS_FIELDS 表中获取所有索引列的信息。

也就是说这4个表是表中之表，那这4个表的元数据去哪里获取呢？没法搞了，只能把这4个表的元数据，就是它们有哪些列、哪些索引等信息硬编码到代码中，然后设计 InnoDB 的大叔又拿出一个固定的页面来记录这4个表的聚簇索引和二级索引对应的 B+树 位置，这个页面就是页号为 7 的页面，类型为 SYS ，记录了 Data Dictionary Header ，也就是数据字典的头部信息。除了这4个表的5个索引的根页面信息外，这个页号为 7 的页面还记录了整个InnoDB存储引擎的一些全局属性，说话太啰嗦，直接看这个页面的示意图：



可以看到这个页面由下边几个部分组成:

名称	中文名	占用空间大小	简单描述
File Header	文件头部	38 字节	页的一些通用信息
Data Dictionary Header	数据字典头部信息	56 字节	记录一些基本系统表的根页面位置以及InnoDB存储引擎的一些全局信息
Segment Header	段头部信息	10 字节	记录本页面所在段对应的INODE Entry位置信息
Empty Space	尚未使用空间	16272 字节	用于页结构的填充，没啥实际意义
File Trailer	文件尾部	8 字节	校验页是否完整

可以看到这个页面里竟然有 Segment Header 部分，意味着设计InnoDB的大叔把这些有关数据字典的信息当成一个段来分配存储空间，我们就姑且称之为 数据字典段 吧。由于目前我们需要记录的数据字典信息非常少（可以看到 Data Dictionary Header 部分仅占用了56字节），所以该段只有一个碎片页，也就是页号为 7 的这个页。

接下来我们需要细细唠叨一下 Data Dictionary Header 部分的各个字段:

- Max Row ID ：我们说过如果我们不显式的为表定义主键，而且表中也没有 UNIQUE 索引，那么 InnoDB 存储引擎会默认为我们生成一个名为 row_id 的列作为主键。因为它是主键，所以每条记录的 row_id 列的值不能重复。原则上只要一个表中的 row_id 列不重复就可以了，也就是说表a和表b拥有一样的 row_id 列也没啥关系，不过设计InnoDB的大叔只提供了这个 Max Row ID 字段，不论哪个拥有 row_id 列的表插入一条记录时，该记录的 row_id 列的值就是 Max Row ID 对应的值，然后再把 Max Row ID 对应的值加1，也就是说这个 Max Row ID 是全局共享的。
- Max Table ID ：InnoDB存储引擎中的所有的表都对应一个唯一的ID，每次新建一个表时，就会把本字段的值作为该表的ID，然后自增本字段的值。
- Max Index ID ：InnoDB存储引擎中的所有的索引都对应一个唯一的ID，每次新建一个索引时，就会把本字段的值作为该索引的ID，然后自增本字段的值。
- Max Space ID ：InnoDB存储引擎中的所有的表空间都对应一个唯一的ID，每次新建一个表空间时，就会把本字段的值作为该表空间的ID，然后自增本字段的值。
- Mix ID Low(Unused) ：这个字段没啥用，跳过。
- Root of SYS_TABLES clust index ：本字段代表 SYS_TABLES 表聚簇索引的根页面的页号。

- Root of SYS_TABLE_IDS sec index ：本字段代表 SYS_TABLES 表为 ID 列建立的二级索引的根页面的页号。
- Root of SYS_COLUMNS clust index ：本字段代表 SYS_COLUMNS 表聚簇索引的根页面的页号。
- Root of SYS_INDEXES clust index 本字段代表 SYS_INDEXES 表聚簇索引的根页面的页号。
- Root of SYS_FIELDS clust index ：本字段代表 SYS_FIELDS 表聚簇索引的根页面的页号。
- Unused ：这4个字节没用，跳过。

以上就是页号为 7 的页面的全部内容，初次看可能会懵逼（因为有点儿绕），大家多瞅几次。

information_schema系统数据库

需要注意一点的是，用户是不能直接访问 InnoDB 的这些内部系统表的，除非你直接去解析系统表空间对应文件系统上的文件。不过设计InnoDB的大叔考虑到查看这些表的内容可能有助于大家分析问题，所以在系统数据库 information_schema 中提供了一些以 innodb_sys 开头的表：

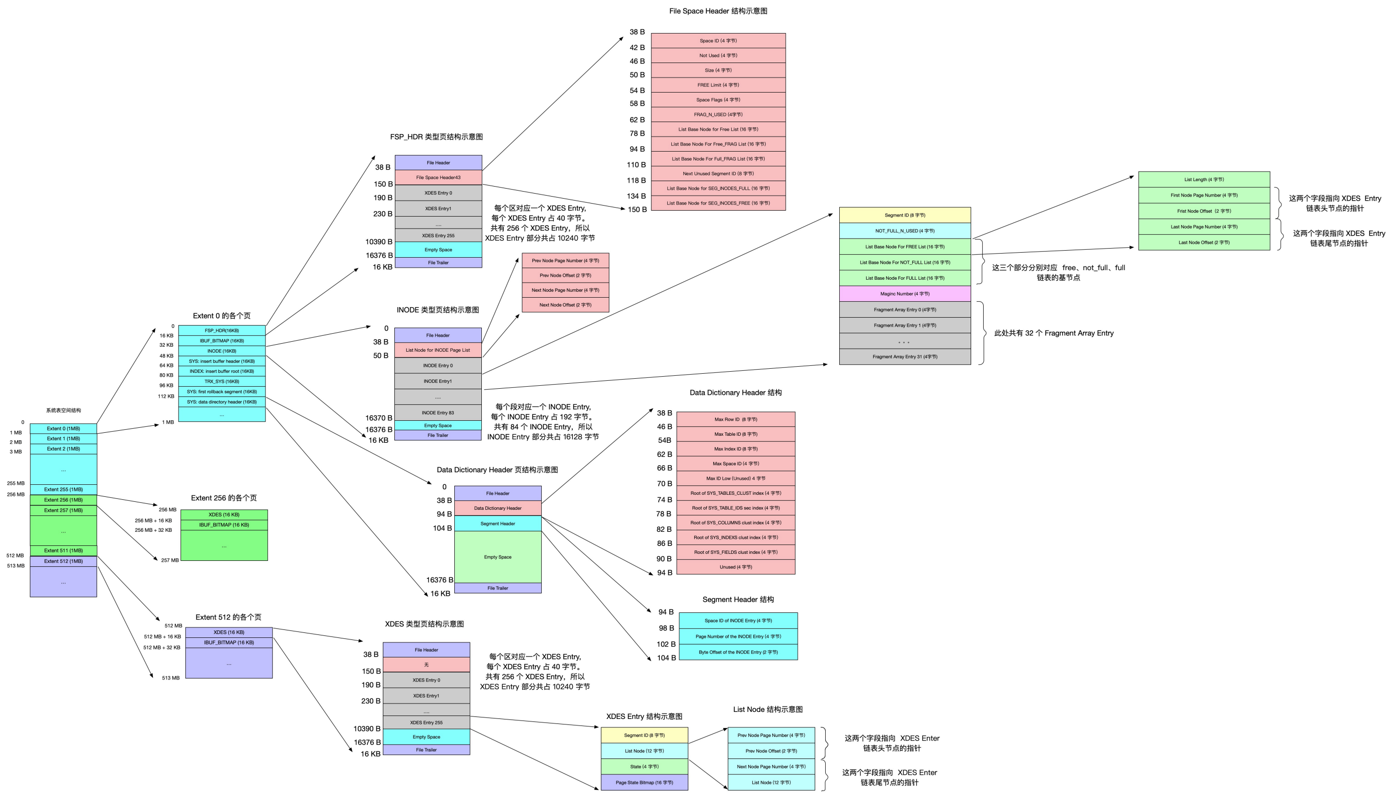
```
mysql> USE information_schema;
Database changed

mysql> SHOW TABLES LIKE 'innodb_sys%';
+--------------------------------------------+
| Tables_in_information_schema (innodb_sys%) |
+--------------------------------------------+
| INNODB_SYS_DATAFILES                       |
| INNODB_SYS_VIRTUAL                         |
| INNODB_SYS_INDEXES                         |
| INNODB_SYS_TABLES                          |
| INNODB_SYS_FIELDS                          |
| INNODB_SYS_TABLESPACES                     |
| INNODB_SYS_FOREIGN_COLS                    |
| INNODB_SYS_COLUMNS                         |
| INNODB_SYS_FOREIGN                         |
| INNODB_SYS_TABLESTATS                      |
+--------------------------------------------+
10 rows in set (0.00 sec)
```

在 information_schema 数据库中的这些以 INNODB_SYS 开头的表并不是真正的内部系统表（内部系统表就是我们上边唠叨的以 SYS 开头的那些表），而是在存储引擎启动时读取这些以 SYS 开头的系统表，然后填充到这些以 INNODB_SYS 开头的表中。以 INNODB_SYS 开头的表和以 SYS 开头的表中的字段并不完全一样，但供大家参考已经足矣。这些表太多了，我就不唠叨了，大家自个儿动手试着查一查这些表中的数据吧哈~

9.3.2 总结图

小册微信交流群2群中一个昵称为 think 同学非常有心的为表空间画了一个全局图，希望能对各位有帮助（这种学习态度实在让我感动😹）：



10 第10章 条条大路通罗马-单表访问方法

标签： MySQL是怎样运行的

对于我们这些 `MySQL` 的使用者来说， `MySQL` 其实就是一个软件，平时用的最多的就是查询功能。DBA时不时丢过来一些慢查询语句让优化，我们如果连查询是怎么执行的都不清楚还优化个毛线，所以是时候掌握真正的技术了。我们在第一章的时候就曾说过， `MySQL Server` 有一个称为 `查询优化器` 的模块，一条查询语句进行语法解析之后就会被交给查询优化器来进行优化，优化的结果就是生成一个所谓的 `执行计划` ，这个执行计划表明了应该使用哪些索引进行查询，表之间的连接顺序是啥样的，最后会按照执行计划中的步骤调用存储引擎提供的方法来真正的执行查询，并将查询结果返回给用户。不过查询优化这个主题有点儿大，在学会跑之前还得先学会走，所以本章先来瞅瞅 `MySQL` 怎么执行单表查询（就是 `FROM` 子句后边只有一个表，最简单的那种查询～）。不过需要强调的一点是，在学习本章前务必看过前边关于记录结构、数据页结构以及索引的部分，如果你不能保证这些东西已经完全掌握，那么本章不适合你。

为了故事的顺利发展，我们先得有个表：

```
CREATE TABLE single_table (
    id INT NOT NULL AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1),
    UNIQUE KEY idx_key2 (key2),
    KEY idx_key3 (key3),
    KEY idx_key_part(key_part1, key_part2, key_part3)
) Engine=InnoDB CHARSET=utf8;
```

我们为这个 single_table 表建立了1个聚簇索引和4个二级索引，分别是：

- 为 id 列建立的聚簇索引。
- 为 key1 列建立的 idx_key1 二级索引。
- 为 key2 列建立的 idx_key2 二级索引，而且该索引是唯一二级索引。
- 为 key3 列建立的 idx_key3 二级索引。
- 为 key_part1 、 key_part2 、 key_part3 列建立的 idx_key_part 二级索引，这也是一个联合索引。

然后我们需要为这个表插入10000行记录，除 id 列外其余的列都插入随机值就好了，具体的插入语句我就不写了，自己写个程序插入吧（id列是自增主键列，不需要我们手动插入）。

10.1 访问方法（access method）的概念

想必各位都用过高德地图来查找到某个地方的路线吧（此处没有为高德地图打广告的意思，他们没给我钱，大家用百度地图也可以啊），如果我们搜西安钟楼到大雁塔之间的路线的话，地图软件会给出n种路线供我们选择，如果我们实在闲的没事儿干并且足够有钱的话，还可以用南辕北辙的方式绕地球一圈到达目的地。也就是说，不论采用哪一种方式，我们最终的目标就是到达大雁塔这个地方。回到 MySQL 中来，我们平时所写的那些查询语句本质上只是一种声明式的语法，只是告诉 MySQL 我们要获取的数据符合哪些规则，至于 MySQL 背地里是怎么把查询结果搞出来的那是 MySQL 自己的事儿。对于单个表的查询来说，设计MySQL的大叔把查询的执行方式大致分为下边两种：

- 使用全表扫描进行查询

  这种执行方式很好理解，就是把表的每一行记录都扫一遍嘛，把符合搜索条件的记录加入到结果集就完了。不管是啥查询都可以使用这种方式执行，当然，这种也是最笨的执行方式。
- 使用索引进行查询

  因为直接使用全表扫描的方式执行查询要遍历好多记录，所以代价可能太大了。如果查询语句中的搜索条件可以使用到某个索引，那直接使用索引来执行查询可能会加快查询执行的时间。使用索引来执行查询的方式五花八门，又可以细分为许多种类：
  - 针对主键或唯一二级索引的等值查询
  - 针对普通二级索引的等值查询
  - 针对索引列的范围查询
  - 直接扫描整个索引

设计 MySQL 的大叔把 MySQL 执行查询语句的方式称之为 访问方法 或者 访问类型 。同一个查询语句可能可以使用多种不同的访问方法来执行，虽然最后的查询结果都是一样的，但是执行的时间可能差老鼻子远了，就像是从钟楼到大雁塔，你可以坐火箭去，也可以坐飞机去，当然也可以坐乌龟去。下边细细道来各种 访问方法 的具体

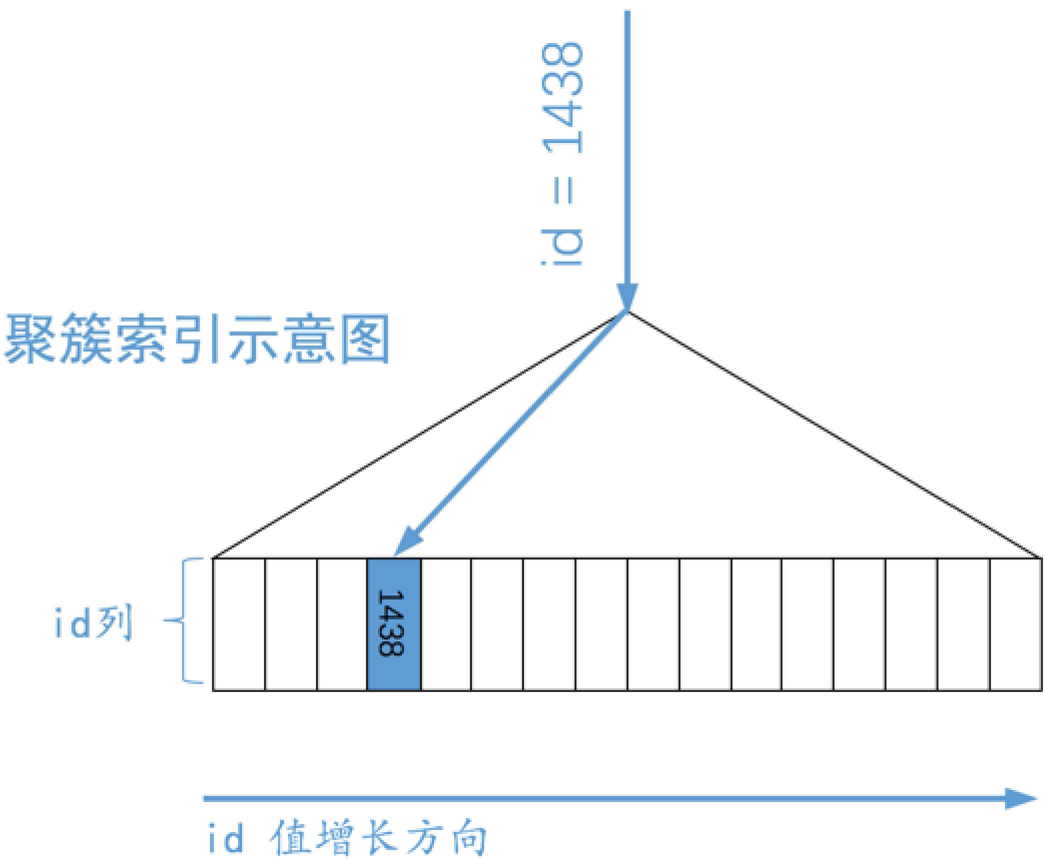
内容。

10.2 const

有的时候我们可以通过主键列来定位一条记录，比方说这个查询：

```
SELECT * FROM single_table WHERE id = 1438;
```

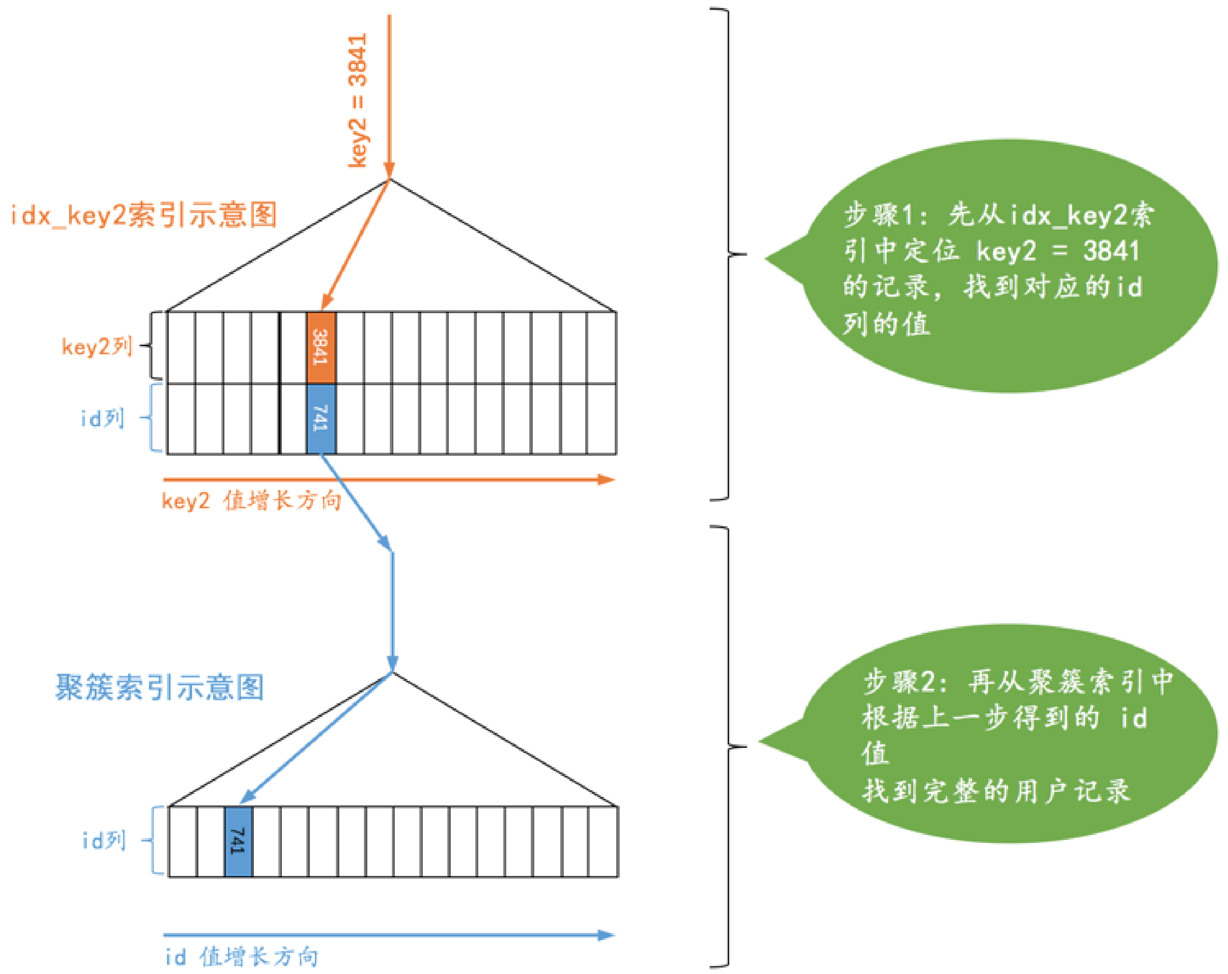
MySQL 会直接利用主键值在聚簇索引中定位对应的用户记录，就像这样：



原谅我把聚簇索引对应的复杂的 B+ 树结构搞了一个极度精简版，为了突出重点，我们忽略掉了 页 的结构，直接把所有的叶子节点的记录都放在一起展示，而且记录中只展示我们关心的索引列，对于 single_table 表的聚簇索引来说，展示的就是 id 列。我们想突出的重点就是： B+ 树叶子节点中的记录是按照索引列排序的，对于的聚簇索引来说，它对应的 B+ 树叶子节点中的记录就是按照 id 列排序的。 B+ 树本来就是一个矮矮的大胖子，所以这样根据主键值定位一条记录的速度贼快。类似的，我们根据唯一二级索引列来定位一条记录的速度也是贼快的，比如下边这个查询：

```
SELECT * FROM single_table WHERE key2 = 3841;
```

这个查询的执行过程的示意图就是这样：



可以看到这个查询的执行分两步，第一步先从 idx_key2 对应的 B+ 树索引中根据 key2 列与常数的等值比较条件定位到一条二级索引记录，然后再根据该记录的 id 值到聚簇索引中获取到完整的用户记录。

设计 MySQL 的大叔认为通过主键或者唯一二级索引列与常数的等值比较来定位一条记录是像坐火箭一样快的，所以他们把这种通过主键或者唯一二级索引列来定位一条记录的访问方法定义为： const ，意思是常数级别的，代价是可以忽略不计的。不过这种 const 访问方法只能在主键列或者唯一二级索引列和一个常数进行等值比较时才有效，如果主键或者唯一二级索引是由多个列构成的话，索引中的每一个列都需要与常数进行等值比较，这个 const 访问方法才有效（这是因为只有该索引中全部列都采用等值比较才可以定位唯一的一条记录）。

对于唯一二级索引来说，查询该列为 NULL 值的情况比较特殊，比如这样：

```
SELECT * FROM single_table WHERE key2 IS NULL;
```

因为唯一二级索引列并不限制 NULL 值的数量，所以上述语句可能访问到多条记录，也就是说 上边这个语句不可以使用 const 访问方法来执行（至于是什么访问方法我们下边马上说）。

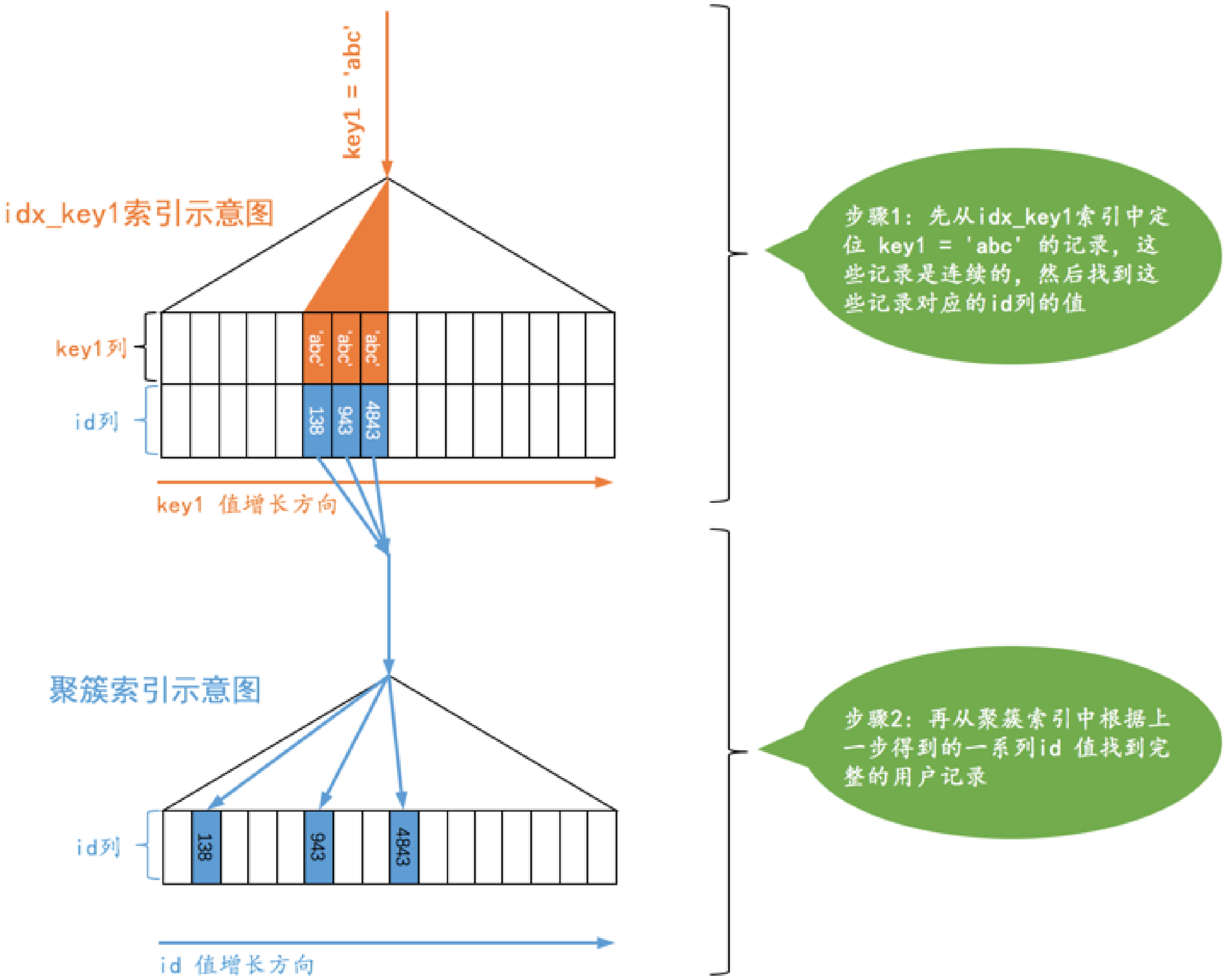
10.3 ref

有时候我们对某个普通的二级索引列与常数进行等值比较，比如这样：

```
SELECT * FROM single_table WHERE key1 = 'abc';
```

对于这个查询，我们当然可以选择全表扫描来逐一对比搜索条件是否满足要求，我们也可以先使用二级索引找到对应记录的 id 值，然后再回表到聚簇索引中查找完整的用户记录。由于普通二级索引并不限制索引列值的唯一性，所以可能找到多条对应的记录，也就是说使用二级索引来执行查询的代价取决于等值匹配到的二级索引记录

条数。如果匹配的记录较少，则回表的代价还是比较低的，所以 MySQL 可能选择使用索引而不是全表扫描的方式来执行查询。设计 MySQL 的大叔就把这种搜索条件为二级索引列与常数等值比较，采用二级索引来执行查询的访问方法称为： ref 。我们看一下采用 ref 访问方法执行查询的图示：



从图示中可以看出，对于普通的二级索引来说，通过索引列进行等值比较后可能匹配到多条连续的记录，而不是像主键或者唯一二级索引那样最多只能匹配1条记录，所以这种 ref 访问方法比 const 差了那么一丢丢，但是在二级索引等值比较时匹配的记录数较少时的效率还是很高的（如果匹配的二级索引记录太多那么回表的成本就太大了），跟坐高铁差不多。不过需要注意下边两种情况：

- 二级索引列值为 NULL 的情况

  不论是普通的二级索引，还是唯一二级索引，它们的索引列对包含 NULL 值的数量并不限制，所以我们采用 key IS NULL 这种形式的搜索条件最多只能使用 ref 的访问方法，而不是 const 的访问方法。
- 对于某个包含多个索引列的二级索引来说，只要是最左边的连续索引列是与常数的等值比较就可能采用 ref 的访问方法，比方说下边这几个查询：

```
SELECT * FROM single_table WHERE key_part1 = 'god like';

SELECT * FROM single_table WHERE key_part1 = 'god like' AND key_part2 = 'legendary';

SELECT * FROM single_table WHERE key_part1 = 'god like' AND key_part2 = 'legendary'
AND key_part3 = 'penta kill';
```

但是如果最左边的连续索引列并不全部是等值比较的话，它的访问方法就不能称为 ref 了，比方说这样：

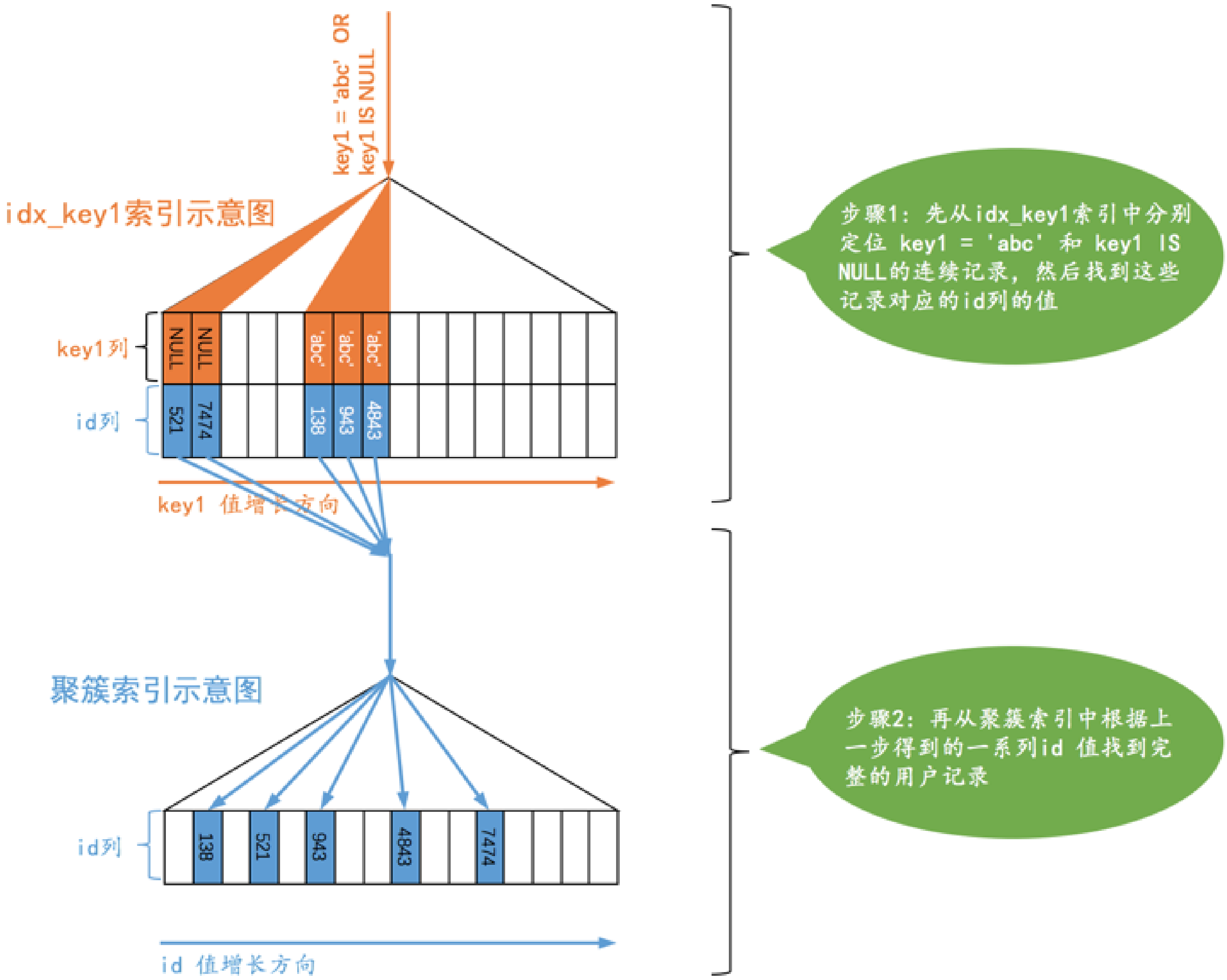
```
SELECT * FROM single_table WHERE key_part1 = 'god like' AND key_part2 > 'legendary';
```

10.4 ref_or_null

有时候我们不仅想找出某个二级索引列的值等于某个常数的记录，还想把该列的值为 NULL 的记录也找出来，就像下边这个查询：

```
SELECT * FROM single_demo WHERE key1 = 'abc' OR key1 IS NULL;
```

当使用二级索引而不是全表扫描的方式执行该查询时，这种类型的查询使用的访问方法就称为 ref_or_null ，这个 ref_or_null 访问方法的执行过程如下：



可以看到，上边的查询相当于先分别从 idx_key1 索引对应的 B+ 树中找出 key1 IS NULL 和 key1 = 'abc' 的两个连续的记录范围，然后根据这些二级索引记录中的 id 值再回表查找完整的用户记录。

10.5 range

我们之前介绍的几种访问方法都是在对索引列与某一个常数进行等值比较的时候才可能使用到（ ref_or_null 比较奇特，还计算了值为 NULL 的情况），但是有时候我们面对的搜索条件更复杂，比如下边这个查询：

```
SELECT * FROM single_table WHERE key2 IN (1438, 6328) OR (key2 >= 38 AND key2 <= 79);
```

我们当然还可以使用全表扫描的方式来执行这个查询，不过也可以使用 二级索引 + 回表 的方式执行，如果采用 二级索引 + 回表 的方式来执行的话，那么此时的搜索条件就不只是要求索引列与常数的等值匹配了，而是索引列需要匹配某个或某些范围的值，在本查询中 key2 列的值只要匹配下列3个范围中的任何一个就算是匹配成功

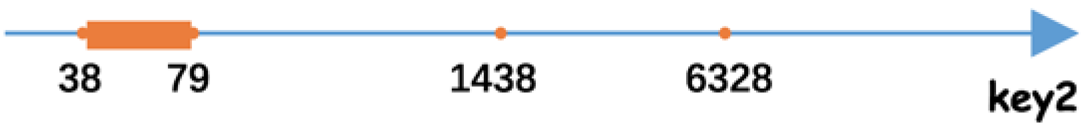
了：

- key2 的值是 1438
- key2 的值是 6328
- key2 的值在 38 和 79 之间。

设计 MySQL 的大叔把这种利用索引进行范围匹配的访问方法称之为： range 。

> 小贴士：
> 此处所说的使用索引进行范围匹配中的 `索引` 可以是聚簇索引，也可以是二级索引。

如果把这几个所谓的 key2 列的值需要满足的 范围 在数轴上体现出来的话，那应该是这个样子：



也就是从数学的角度看，每一个所谓的范围都是数轴上的一个 区间 ，3个范围也就对应着3个区间：

- 范围1： key2 = 1438
- 范围2： key2 = 6328
- 范围3： key2 ∈ [38, 79] ，注意这里是闭区间。

我们可以把那种索引列等值匹配的情况称之为 单点区间 ，上边所说的 范围1 和 范围2 都可以被称为单点区间，像 范围3 这种的我们可以称为连续范围区间。

10.6 index

看下边这个查询：

```
SELECT key_part1, key_part2, key_part3 FROM single_table WHERE key_part2 = 'abc';
```

由于 key_part2 并不是联合索引 idx_key_part 最左索引列，所以我们无法使用 ref 或者 range 访问方法来执行这个语句。但是这个查询符合下边这两个条件：

- 它的查询列表只有3个列： key_part1 ， key_part2 ， key_part3 ，而索引 idx_key_part 又包含这三个列。
- 搜索条件中只有 key_part2 列。这个列也包含在索引 idx_key_part 中。

也就是说我们可以直接通过遍历 idx_key_part 索引的叶子节点的记录来比较 key_part2 = 'abc' 这个条件是否成立，把匹配成功的二级索引记录的 key_part1 ， key_part2 ， key_part3 列的值直接加到结果集中就行了。由于二级索引记录比聚簇索记录小的多（聚簇索引记录要存储所有用户定义的列以及所谓的隐藏列，而二级索引记录只需要存放索引列和主键），而且这个过程也不用进行回表操作，所以直接遍历二级索引比直接遍历聚簇索引的成本要小很多，设计 MySQL 的大叔就把这种采用遍历二级索引记录的执行方式称之为： index 。

10.7 all

最直接的查询执行方式就是我们已经提了无数遍的全表扫描，对于 InnoDB 表来说也就是直接扫描聚簇索引，设计 MySQL 的大叔把这种使用全表扫描执行查询的方式称之为： all 。

10.8 注意事项

10.8.1 重温 二级索引 + 回表

一般情况下只能利用单个二级索引执行查询，比方说下边的这个查询：

```
SELECT * FROM single_table WHERE key1 = 'abc' AND key2 > 1000;
```

查询优化器会识别到这个查询中的两个搜索条件：

- key1 = 'abc'
- key2 > 1000

优化器一般会根据 single_table 表的统计数据来判断到底使用哪个条件到对应的二级索引中查询扫描的行数会更少，选择那个扫描行数较少的条件到对应的二级索引中查询（关于如何比较的细节我们后边的章节中会唠叨）。然后将从该二级索引中查询到的结果经过回表得到完整的用户记录后再根据其余的 WHERE 条件过滤记录。一般来说，等值查找比范围查找需要扫描的行数更少（也就是 ref 的访问方法一般比 range 好，但这也不总是一定的，也可能采用 ref 访问方法的那个索引列的值为特定值的行数特别多），所以这里假设优化器决定使用 idx_key1 索引进行查询，那么整个查询过程可以分为两个步骤：

- 步骤1：使用二级索引定位记录的阶段，也就是根据条件 key1 = 'abc' 从 idx_key1 索引代表的 B+ 树中找到对应的二级索引记录。
- 步骤2：回表阶段，也就是根据上一步骤中找到的记录的主键值进行 回表 操作，也就是到聚簇索引中找到对应的完整的用户记录，再根据条件 key2 > 1000 到完整的用户记录继续过滤。将最终符合过滤条件的记录返回给用户。

这里需要特别提醒大家的一点是，因为二级索引的节点中的记录只包含索引列和主键，所以在步骤1中使用 idx_key1 索引进行查询时只会用到与 key1 列有关的搜索条件，其余条件，比如 key2 > 1000 这个条件在步骤1中是用不到的，只有在步骤2完成回表操作后才能继续针对完整的用户记录中继续过滤。

> 小贴士：
> 需要注意的是，我们说一般情况下执行一个查询只会用到单个二级索引，不过还是有特殊情况的，我们后边会详细唠叨的。

10.8.2 明确range访问方法使用的范围区间

其实对于 B+ 树索引来说，只要索引列和常数使用 = 、 <=> 、 IN 、 NOT IN 、 IS NULL 、 IS NOT NULL 、 > 、 < 、 >= 、 <= 、 BETWEEN 、 != （不等于也可以写成 <> ）或者 LIKE 操作符连接起来，就可以产生一个所谓的 区间 。

> 小贴士：
> LIKE操作符比较特殊，只有在匹配完整字符串或者匹配字符串前缀时才可以利用索引，具体原因我们在前边的章节中唠叨过了，这里就不赘述了。
> IN操作符的效果和若干个等值匹配操作符`=`之间用`OR`连接起来是一样的，也就是说会产生多个单点区间，比如下边这两个语句的效果是一样的：
> SELECT * FROM single_table WHERE key2 IN (1438, 6328);
> SELECT * FROM single_table WHERE key2 = 1438 OR key2 = 6328;

不过在日常的工作中，一个查询的 WHERE 子句可能有很多个小的搜索条件，这些搜索条件需要使用 AND 或者 OR 操作符连接起来，虽然大家都知道这两个操作符的作用，但我还是要再说一遍：

- cond1 AND cond2 ：只有当 cond1 和 cond2 都为 TRUE 时整个表达式才为 TRUE 。
- cond1 OR cond2 ：只要 cond1 或者 cond2 中有一个为 TRUE 整个表达式就为 TRUE 。

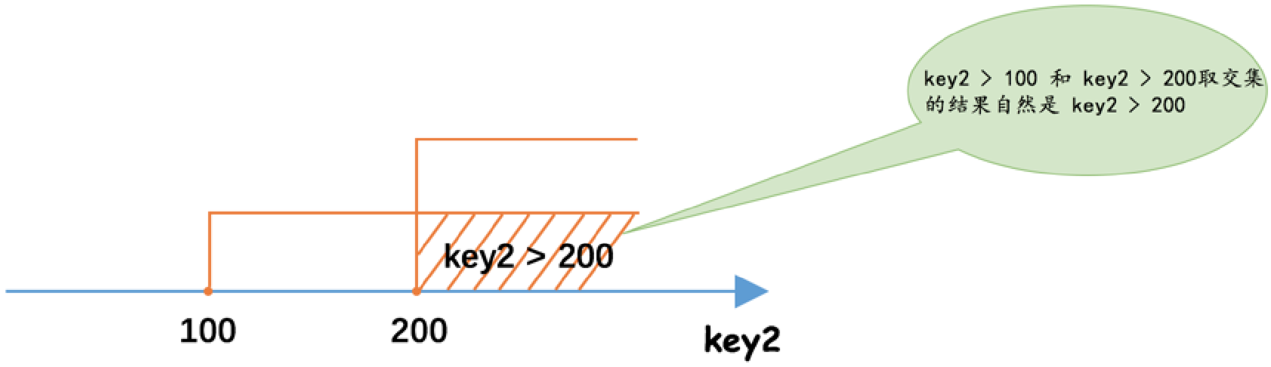
当我们想使用 range 访问方法来执行一个查询语句时，重点就是找出该查询可用的索引以及这些索引对应的范围区间。下边分两种情况看一下怎么从由 AND 或 OR 组成的复杂搜索条件中提取出正确的范围区间。

10.8.2.1 所有搜索条件都可以使用某个索引的情况

有时候每个搜索条件都可以使用到某个索引，比如下边这个查询语句：

```
SELECT * FROM single_table WHERE key2 > 100 AND key2 > 200;
```

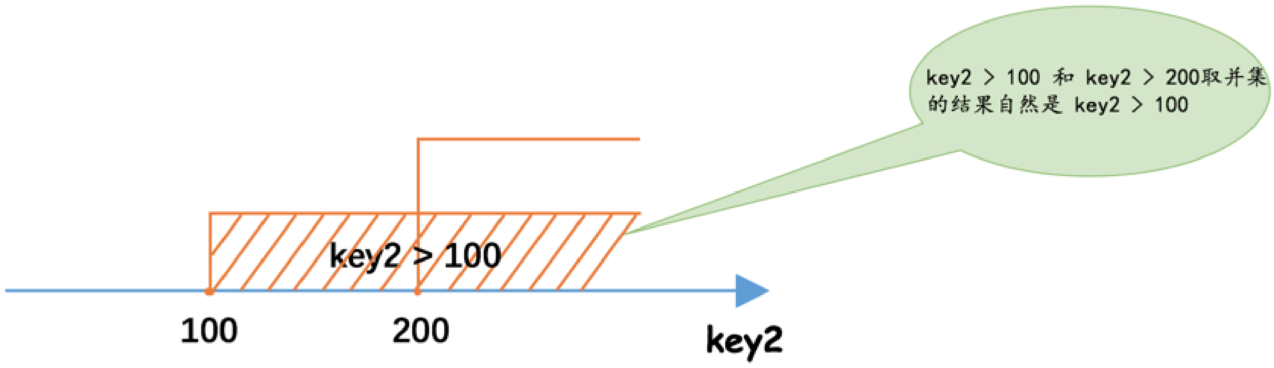
这个查询中的搜索条件都可以使用到 key2 ，也就是说每个搜索条件都对应着一个 idx_key2 的范围区间。这两个小的搜索条件使用 AND 连接起来，也就是要取两个范围区间的交集，在我们使用 range 访问方法执行查询时，使用的 idx_key2 索引的范围区间的确定过程就如下图所示：



key2 > 100 和 key2 > 200 交集当然就是 key2 > 200 了，也就是说上边这个查询使用 idx_key2 的范围区间就是 (200, +∞) 。这东西小学都学过吧，再不济初中肯定都学过。我们再看一下使用 OR 将多个搜索条件连接在一起的情况：

```
SELECT * FROM single_table WHERE key2 > 100 OR key2 > 200;
```

OR 意味着需要取各个范围区间的并集，所以上边这个查询在我们使用 range 访问方法执行查询时，使用的 idx_key2 索引的范围区间的确定过程就如下图所示：



也就是说上边这个查询使用 idx_key2 的范围区间就是 (100, +∞) 。

10.8.2.2 有的搜索条件无法使用索引的情况

比如下边这个查询：

```
SELECT * FROM single_table WHERE key2 > 100 AND common_field = 'abc';
```

请注意，这个查询语句中能利用的索引只有 idx_key2 一个，而 idx_key2 这个二级索引的记录中又不包含 common_field 这个字段，所以在使用二级索引 idx_key2 定位记录的阶段用不到 common_field = 'abc' 这个条件，这个条件是在回表获取了完整的用户记录后才使用的，而 范围区间 是为了到索引中取记录中提出的概念，

所以在确定 范围区间 的时候不需要考虑 common_field = 'abc' 这个条件，我们在为某个索引确定范围区间的时候只需要把用不到相关索引的搜索条件替换为 TRUE 就好了。

> 小贴士：
> 之所以把用不到索引的搜索条件替换为TRUE，是因为我们不打算使用这些条件进行在该索引上进行过滤，所以不管索引的记录满不满足这些条件，我们都把它们选取出来，待到之后回表的时候再使用它们过滤。

我们把上边的查询中用不到 idx_key2 的搜索条件替换后就是这样：

```
SELECT * FROM single_table WHERE key2 > 100 AND TRUE;
```

化简之后就是这样：

```
SELECT * FROM single_table WHERE key2 > 100;
```

也就是说最上边那个查询使用 idx_key2 的范围区间就是：(100, +∞)。

再来看一下使用 OR 的情况：

```
SELECT * FROM single_table WHERE key2 > 100 OR common_field = 'abc';
```

同理，我们把使用不到 idx_key2 索引的搜索条件替换为 TRUE ：

```
SELECT * FROM single_table WHERE key2 > 100 OR TRUE;
```

接着化简：

```
SELECT * FROM single_table WHERE TRUE;
```

额，这也就说说明如果我们强制使用 idx_key2 执行查询的话，对应的范围区间就是 (-∞, +∞)，也就是需要将全部二级索引的记录进行回表，这个代价肯定比直接全表扫描都大了。也就是说一个使用到索引的搜索条件和没有使用该索引的搜索条件使用 OR 连接起来后是无法使用该索引的。

10.8.2.3 复杂搜索条件下找出范围匹配的区间

有的查询的搜索条件可能特别复杂，光是找出范围匹配的各个区间就挺烦的，比方说下边这个：

```
SELECT * FROM single_table WHERE
        (key1 > 'xyz' AND key2 = 748 ) OR
        (key1 < 'abc' AND key1 > 'lmn') OR
        (key1 LIKE '%suf' AND key1 > 'zzz' AND (key2 < 8000 OR common_field = 'abc')) ;
```

我滴个神，这个搜索条件真是绝了，不过大家不要被复杂的表象迷住了双眼，按着下边这个套路分析一下：

- 首先查看 WHERE 子句中的搜索条件都涉及到了哪些列，哪些列可能使用到索引。

  这个查询的搜索条件涉及到了 key1 、 key2 、 common_field 这3个列，然后 key1 列有普通的二级索引 idx_key1 ， key2 列有唯一二级索引 idx_key2 。
- 对于那些可能用到的索引，分析它们的范围区间。
  - 假设我们使用 idx_key1 执行查询
    - 我们需要把那些用不到该索引的搜索条件暂时移除掉，移除方法也简单，直接把它们替换为 TRUE 就好了。上边的查询中除了有关 key2 和 common_field 列不能使用到 idx_key1 索引外， key1 LIKE '%suf' 也使用不到索引，所以把这些搜索条件替换为 TRUE 之后的样子就是这样：

```
(key1 > 'xyz' AND TRUE ) OR
(key1 < 'abc' AND key1 > 'lmn') OR
(TRUE AND key1 > 'zzz' AND (TRUE OR TRUE))
```

化简一下上边的搜索条件就是下边这样：

```
(key1 > 'xyz') OR
(key1 < 'abc' AND key1 > 'lmn') OR
(key1 > 'zzz')
```

- 替换掉永远为 `TRUE` 或 `FALSE` 的条件

  因为符合 `key1 < 'abc' AND key1 > 'lmn'` 永远为 `FALSE`，所以上边的搜索条件可以被写成这样：

  ```
  (key1 > 'xyz') OR (key1 > 'zzz')
  ```

- 继续化简区间

  `key1 > 'xyz'` 和 `key1 > 'zzz'` 之间使用 `OR` 操作符连接起来的，意味着要取并集，所以最终的结果化简的到的区间就是：`key1 > xyz`。也就是说：上边那个有一坨搜索条件的查询语句如果使用 idx_key1 索引执行查询的话，需要把满足 `key1 > xyz` 的二级索引记录都取出来，然后拿着这些记录的id再进行回表，得到完整的用户记录之后再使用其他的搜索条件进行过滤。

▪ 假设我们使用 `idx_key2` 执行查询

- 我们需要把那些用不到该索引的搜索条件暂时使用 `TRUE` 条件替换掉，其中有关 `key1` 和 `common_field` 的搜索条件都需要被替换掉，替换结果就是：

  ```
  (TRUE AND key2 = 748 ) OR
  (TRUE AND TRUE) OR
  (TRUE AND TRUE AND (key2 < 8000 OR TRUE))
  ```

  哎呀呀，`key2 < 8000 OR TRUE` 的结果肯定是 `TRUE` 呀，也就是说化简之后的搜索条件成这样了：

  ```
  key2 = 748 OR TRUE
  ```

  这个化简之后的结果就更简单了：

  ```
  TRUE
  ```

  这个结果也就意味着如果我们要使用 `idx_key2` 索引执行查询语句的话，需要扫描 `idx_key2` 二级索引的所有记录，然后再回表，这不是得不偿失么，所以这种情况下不会使用 `idx_key2` 索引的。

10.8.3 索引合并

我们前边说过 `MySQL` 在一般情况下执行一个查询时最多只会用到单个二级索引，但不是还有特殊情况么，在这些特殊情况下也可能在一个查询中使用到多个二级索引，设计 `MySQL` 的大叔把这种使用到多个索引来完成一次查询的执行方法称之为：`index merge`，具体的索引合并算法有下边三种。

10.8.3.1 Intersection合并

`Intersection` 翻译过来的意思是 `交集`。这里是说某个查询可以使用多个二级索引，将从多个二级索引中查询到的结果取交集，比方说下边这个查询：

```
SELECT * FROM single_table WHERE key1 = 'a' AND key3 = 'b';
```

假设这个查询使用 `Intersection` 合并的方式执行的话，那这个过程就是这样的：

- 从 `idx_key1` 二级索引对应的 `B+` 树中取出 `key1 = 'a'` 的相关记录。

- 从 idx_key3 二级索引对应的 B+ 树中取出 key3 = 'b' 的相关记录。
- 二级索引的记录都是由 索引列 + 主键 构成的，所以我们可以计算出这两个结果集中 id 值的交集。
- 按照上一步生成的 id 值列表进行回表操作，也就是从聚簇索引中把指定 id 值的完整用户记录取出来，返回给用户。

这里有同学会思考：为啥不直接使用 idx_key1 或者 idx_key3 只根据某个搜索条件去读取一个二级索引，然后回表后再过滤另外一个搜索条件呢？这里要分析一下两种查询执行方式之间需要的成本代价。

只读取一个二级索引的成本：

- 按照某个搜索条件读取一个二级索引
- 根据从该二级索引得到的主键值进行回表操作，然后再过滤其他的搜索条件

读取多个二级索引之后取交集成本：

- 按照不同的搜索条件分别读取不同的二级索引
- 将从多个二级索引得到的主键值取交集，然后进行回表操作

虽然读取多个二级索引比读取一个二级索引消耗性能，但是读取二级索引的操作是 顺序I/O ，而回表操作是 随机I/O ，所以如果只读取一个二级索引时需要回表的记录数特别多，而读取多个二级索引之后取交集的记录数非常少，当节省的因为 回表 而造成的性能损耗比访问多个二级索引带来的性能损耗更高时，读取多个二级索引后取交集比只读取一个二级索引的成本更低。

MySQL 在某些特定的情况下才可能会使用到 Intersection 索引合并：

- 情况一：二级索引列是等值匹配的情况，对于联合索引来说，在联合索引中的每个列都必须等值匹配，不能出现只出现匹配部分列的情况。

  比方说下边这个查询可能用到 idx_key1 和 idx_key_part 这两个二级索引进行 Intersection 索引合并的操作：

  ```
  SELECT * FROM single_table WHERE key1 = 'a' AND key_part1 = 'a' AND key_part2 = 'b'
  AND key_part3 = 'c';
  ```

  而下边这两个查询就不能进行 Intersection 索引合并：

  ```
  SELECT * FROM single_table WHERE key1 > 'a' AND key_part1 = 'a' AND key_part2 = 'b'
  AND key_part3 = 'c';

  SELECT * FROM single_table WHERE key1 = 'a' AND key_part1 = 'a';
  ```

  第一个查询是因为对 key1 进行了范围匹配，第二个查询是因为联合索引 idx_key_part 中的 key_part2 列并没有出现在搜索条件中，所以这两个查询不能进行 Intersection 索引合并。
- 情况二：主键列可以是范围匹配

  比方说下边这个查询可能用到主键和 idx_key1 进行 Intersection 索引合并的操作：

  ```
  SELECT * FROM single_table WHERE id > 100 AND key1 = 'a';
  ```

为啥呢？凭啥呀？突然冒出这么两个规定让大家一脸懵逼，下边我们慢慢品一品这里头的玄机。这话还得从 InnoDB 的索引结构说起，你要是记不清麻烦再回头看看。对于 InnoDB 的二级索引来说，记录先是按照索引列进行排序，如果该二级索引是一个联合索引，那么会按照联合索引中的各个列依次排序。而二级索引的用户记录是由 索引列 + 主键 构成的，二级索引列的值相同的记录可能会有好多条，这些索引列的值相同的记录又是按照 主键 的值进行排序的。所以重点来了，之所以在二级索引列都是等值匹配的情况下才可能使用 Intersection 索引合并，是因为只有在这种情况下根据二级索引查询出的结果集是按照主键值排序的。

so？还是没看懂根据二级索引查询出的结果集是按照主键值排序的对使用 Intersection 索引合并有啥好处？小伙子，别忘了 Intersection 索引合并会把从多个二级索引中查询出的主键值求交集，如果从各个二级索引中查询的到的结果集本身就是已经按照主键排好序的，那么求交集的过程就很easy啦。假设某个查询使用 Intersection 索引合并的方式从 idx_key1 和 idx_key2 这两个二级索引中获取到的主键值分别是：

- 从 idx_key1 中获取到已经排好序的主键值：1、3、5
- 从 idx_key2 中获取到已经排好序的主键值：2、3、4

那么求交集的过程就是这样：逐个取出这两个结果集中最小的主键值，如果两个值相等，则加入最后的交集结果中，否则丢弃当前较小的主键值，再取该丢弃的主键值所在结果集的后一个主键值来比较，直到某个结果集中的主键值用完了，如果还是觉得不太明白那继续往下看：

- 先取出这两个结果集中较小的主键值做比较，因为 1 < 2 ，所以把 idx_key1 的结果集的主键值 1 丢弃，取出后边的 3 来比较。
- 因为 3 > 2 ，所以把 idx_key2 的结果集的主键值 2 丢弃，取出后边的 3 来比较。
- 因为 3 = 3 ，所以把 3 加入到最后的交集结果中，继续两个结果集后边的主键值来比较。
- 后边的主键值也不相等，所以最后的交集结果中只包含主键值 3 。

别看我们写的啰嗦，这个过程其实可快了，时间复杂度是 O(n) ，但是如果从各个二级索引中查询出的结果集并不是按照主键排序的话，那就要先把结果集中的主键值排序完再来做上边的那个过程，就比较耗时了。

> 小贴士：
> 按照有序的主键值去回表取记录有个专有名词儿，叫：Rowid Ordered Retrieval，简称ROR，以后大家在某些地方见到这个名词儿就眼熟了。

另外，不仅是多个二级索引之间可以采用 Intersection 索引合并，索引合并也可以有聚簇索引参加，也就是我们上边写的 情况二 ：在搜索条件中有主键的范围匹配的情况下也可以使用 Intersection 索引合并索引合并。为啥主键这就可以范围匹配了？还是得回到应用场景里，比如看下边这个查询：

```
SELECT * FROM single_table WHERE key1 = 'a' AND id > 100;
```

假设这个查询可以采用 Intersection 索引合并，我们理所当然的以为这个查询会分别按照 id > 100 这个条件从聚簇索引中获取一些记录，在通过 key1 = 'a' 这个条件从 idx_key1 二级索引中获取一些记录，然后再求交集，其实这样就把问题复杂化了，没必要从聚簇索引中获取一次记录。别忘了二级索引的记录中都带有主键值的，所以可以在从 idx_key1 中获取到的主键值上直接运用条件 id > 100 过滤就行了，这样多简单。所以涉及主键的搜索条件只不过是为了从别的二级索引得到的结果集中过滤记录罢了，是不是等值匹配不重要。

当然，上边说的 情况一 和 情况二 只是发生 Intersection 索引合并的必要条件，不是充分条件。也就是说即使情况一、情况二成立，也不一定发生 Intersection 索引合并，这得看优化器的心情。优化器只有在单独根据搜索条件从某个二级索引中获取的记录数太多，导致回表开销太大，而通过 Intersection 索引合并后需要回表的记录数大大减少时才会使用 Intersection 索引合并。

10.8.3.2 Union合并

我们在写查询语句时经常想把既符合某个搜索条件的记录取出来，也把符合另外的某个搜索条件的记录取出来，我们说这些不同的搜索条件之间是 OR 关系。有时候 OR 关系的不同搜索条件会使用到不同的索引，比方说这样：

```
SELECT * FROM single_table WHERE key1 = 'a' OR key3 = 'b'
```

Intersection 是交集的意思，这适用于使用不同索引的搜索条件之间使用 AND 连接起来的情况； Union 是并集的意思，适用于使用不同索引的搜索条件之间使用 OR 连接起来的情况。与 Intersection 索引合并类似，MySQL 在某些特定的情况下才可能会使用到 Union 索引合并：

- 情况一：二级索引列是等值匹配的情况，对于联合索引来说，在联合索引中的每个列都必须等值匹配，不能出现只出现匹配部分列的情况。

比方说下边这个查询可能用到 idx_key1 和 idx_key_part 这两个二级索引进行 Union 索引合并的操作：

```
SELECT * FROM single_table WHERE key1 = 'a' OR ( key_part1 = 'a' AND key_part2 = 'b' AND key_part3 = 'c');
```

而下边这两个查询就不能进行 Union 索引合并：

```
SELECT * FROM single_table WHERE key1 > 'a' OR (key_part1 = 'a' AND key_part2 = 'b' AND key_part3 = 'c');

SELECT * FROM single_table WHERE key1 = 'a' OR key_part1 = 'a';
```

第一个查询是因为对 key1 进行了范围匹配，第二个查询是因为联合索引 idx_key_part 中的 key_part2 列并没有出现在搜索条件中，所以这两个查询不能进行 Union 索引合并。

- 情况二：主键列可以是范围匹配
- 情况三：使用 Intersection 索引合并的搜索条件

  这种情况其实也挺好理解，就是搜索条件的某些部分使用 Intersection 索引合并的方式得到的主键集合和其他方式得到的主键集合取交集，比方说这个查询：

  ```
  SELECT * FROM single_table WHERE key_part1 = 'a' AND key_part2 = 'b' AND key_part3 = 'c' OR (key1 = 'a' AND key3 = 'b');
  ```

  优化器可能采用这样的方式来执行这个查询：

  - 先按照搜索条件 key1 = 'a' AND key3 = 'b' 从索引 idx_key1 和 idx_key3 中使用 Intersection 索引合并的方式得到一个主键集合。
  - 再按照搜索条件 key_part1 = 'a' AND key_part2 = 'b' AND key_part3 = 'c' 从联合索引 idx_key_part 中得到另一个主键集合。
  - 采用 Union 索引合并的方式把上述两个主键集合取并集，然后进行回表操作，将结果返回给用户。

当然，查询条件符合了这些情况也不一定就会采用 Union 索引合并，也得看优化器的心情。优化器只有在单独根据搜索条件从某个二级索引中获取的记录数比较少，通过 Union 索引合并后进行访问的代价比全表扫描更小时才会使用 Union 索引合并。

10.8.3.3 Sort-Union合并

Union 索引合并的使用条件太苛刻，必须保证各个二级索引列在进行等值匹配的条件下才可能被用到，比方说下边这个查询就无法使用到 Union 索引合并：

```
SELECT * FROM single_table WHERE key1 < 'a' OR key3 > 'z'
```

这是因为根据 key1 < 'a' 从 idx_key1 索引中获取的二级索引记录的主键值不是排好序的，根据 key3 > 'z' 从 idx_key3 索引中获取的二级索引记录的主键值也不是排好序的，但是 key1 < 'a' 和 key3 > 'z' 这两个条件又特别让我们动心，所以我们可以这样：

- 先根据 key1 < 'a' 条件从 idx_key1 二级索引总获取记录，并按照记录的主键值进行排序
- 再根据 key3 > 'z' 条件从 idx_key3 二级索引总获取记录，并按照记录的主键值进行排序
- 因为上述的两个二级索引主键值都是排好序的，剩下的操作和 Union 索引合并方式就一样了。

我们把上述这种先按照二级索引记录的主键值进行排序，之后按照 Union 索引合并方式执行的方式称之为 Sort-Union 索引合并，很显然，这种 Sort-Union 索引合并比单纯的 Union 索引合并多了一步对二级索引记录的主键值排序的过程。

> 小贴士：
> 为啥有Sort-Union索引合并，就没有Sort-Intersection索引合并么？是的，的确没有Sort-Intersection索引合并这么一说，
> Sort-Union的适用场景是单独根据搜索条件从某个二级索引中获取的记录数比较少，这样即使对这些二级索引记录按照主键值进行排序的成本也不会太高
> 而Intersection索引合并的适用场景是单独根据搜索条件从某个二级索引中获取的记录数太多，导致回表开销太大，合并后可以明显降低回表开销，但是如果加入Sort-Intersection后，就需要为大量的二级索引记录按照主键值进行排序，这个成本可能比回表查询都高了，所以也就没有引入Sort-Intersection这个玩意儿。

10.8.3.4 索引合并注意事项

10.8.3.5 联合索引替代Intersection索引合并

```
SELECT * FROM single_table WHERE key1 = 'a' AND key3 = 'b';
```

这个查询之所以可能使用 `Intersection` 索引合并的方式执行，还不是因为 `idx_key1` 和 `idx_key3` 是两个单独的 `B+` 树索引，你要是把这两个列搞一个联合索引，那直接使用这个联合索引就把事情搞定了，何必用啥索引合并呢，就像这样：

```
ALTER TABLE single_table drop index idx_key1, idx_key3, add index idx_key1_key3(key1, key3);
```

这样我们把没用的 `idx_key1` 、 `idx_key3` 都干掉，再添加一个联合索引 `idx_key1_key3` ，使用这个联合索引进行查询简直是又快又好，既不用多读一棵 `B+` 树，也不用合并结果，何乐而不为？

> 小贴士：
> 不过小心有单独对key3列进行查询的业务场景，这样子不得不再把key3列的单独索引给加上。

11 第11章 两个表的亲密接触-连接的原理

标签： MySQL 是怎样运行的

搞数据库一个避不开的概念就是 `Join` ，翻译成中文就是 `连接` 。相信很多小伙伴在初学连接的时候有些一脸懵逼，理解了连接的语义之后又可能不明白各个表中的记录到底是怎么连起来的，以至于在使用的时候常常陷入下边两种误区：

- 误区一：业务至上，管他三七二十一，再复杂的查询也用在一个连接语句中搞定。
- 误区二：敬而远之，上次 DBA 那给报过来的慢查询就是因为使用了连接导致的，以后再也不敢用了。

所以本章就来扒一扒连接的原理。考虑到一部分小伙伴可能忘了连接是个啥或者压根儿就不知道，为了节省他们百度或者看其他书的宝贵时间以及为了我的书凑字数，我们先来介绍一下 MySQL 中支持的一些连接语法。

11.1 连接简介

11.1.1 连接的本质

为了故事的顺利发展，我们先建立两个简单的表并给它们填充一点数据：

```
mysql> CREATE TABLE t1 (m1 int, n1 char(1));
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2 (m2 int, n2 char(1));
Query OK, 0 rows affected (0.02 sec)

mysql> INSERT INTO t1 VALUES(1, 'a'), (2, 'b'), (3, 'c');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

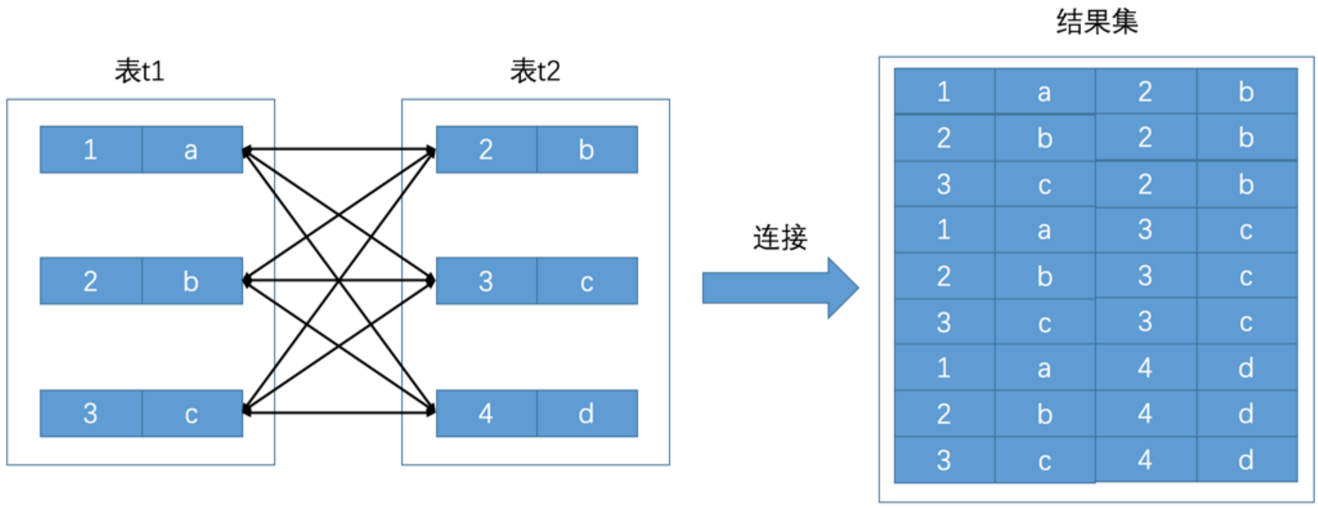
mysql> INSERT INTO t2 VALUES(2, 'b'), (3, 'c'), (4, 'd');
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

我们成功建立了 t1 、 t2 两个表，这两个表都有两个列，一个是 INT 类型的，一个是 CHAR(1) 类型的，填充好数据的两个表长这样：

```
mysql> SELECT * FROM t1;
+------+------+
| m1   | n1   |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM t2;
+------+------+
| m2   | n2   |
+------+------+
|    2 | b    |
|    3 | c    |
|    4 | d    |
+------+------+
3 rows in set (0.00 sec)
```

连接 的本质就是把各个连接表中的记录都取出来依次匹配的组合加入结果集并返回给用户。所以我们把 t1 和 t2 两个表连接起来的过程如下图所示：



这个过程看起来就是把 t1 表的记录和 t2 的记录连起来组成新的更大的记录，所以这个查询过程称之为连接查询。连接查询的结果集中包含一个表中的每一条记录与另一个表中的每一条记录相互匹配的组合，像这样的结果集就可以称之为 笛卡尔积 。因为表 t1 中有3条记录，表 t2 中也有3条记录，所以这两个表连接之后的笛卡尔积就有 3×3=9 行记录。在 MySQL 中，连接查询的语法也很随意，只要在 FROM 语句后边跟多个表名就好了，比如我们把 t1 表和 t2 表连接起来的查询语句可以写成这样：

```
mysql> SELECT * FROM t1, t2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    1 | a    |    2 | b    |
|    2 | b    |    2 | b    |
|    3 | c    |    2 | b    |
|    1 | a    |    3 | c    |
|    2 | b    |    3 | c    |
|    3 | c    |    3 | c    |
|    1 | a    |    4 | d    |
|    2 | b    |    4 | d    |
|    3 | c    |    4 | d    |
+------+------+------+------+
9 rows in set (0.00 sec)
```

11.1.2 连接过程简介

如果我们乐意，我们可以连接任意数量张表，但是如果没有任何限制条件的话，这些表连接起来产生的 笛卡尔积 可能是非常巨大的。比方说3个100行记录的表连接起来产生的 笛卡尔积 就有 100×100×100=1000000 行数据！所以在连接的时候过滤掉特定记录组合是有必要的，在连接查询中的过滤条件可以分成两种：

- 涉及单表的条件

  这种只设计单表的过滤条件我们之前都提到过一万遍了，我们之前也一直称为 搜索条件 ，比如 t1.m1 > 1 是只针对 t1 表的过滤条件， t2.n2 < 'd' 是只针对 t2 表的过滤条件。
- 涉及两表的条件

  这种过滤条件我们之前没见过，比如 t1.m1 = t2.m2 、 t1.n1 > t2.n2 等，这些条件中涉及到了两个表，我们稍后会仔细分析这种过滤条件是如何使用的哈。

下边我们就要看一下携带过滤条件的连接查询的大致执行过程了，比方说下边这个查询语句：

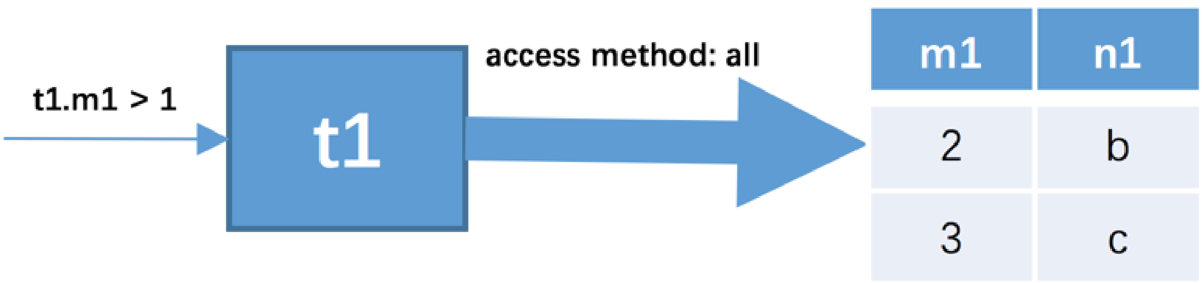
```
SELECT * FROM t1, t2 WHERE t1.m1 > 1 AND t1.m1 = t2.m2 AND t2.n2 < 'd';
```

在这个查询中我们指明了这三个过滤条件：

- `t1.m1 > 1`
- `t1.m1 = t2.m2`
- `t2.n2 < 'd'`

那么这个连接查询的大致执行过程如下：

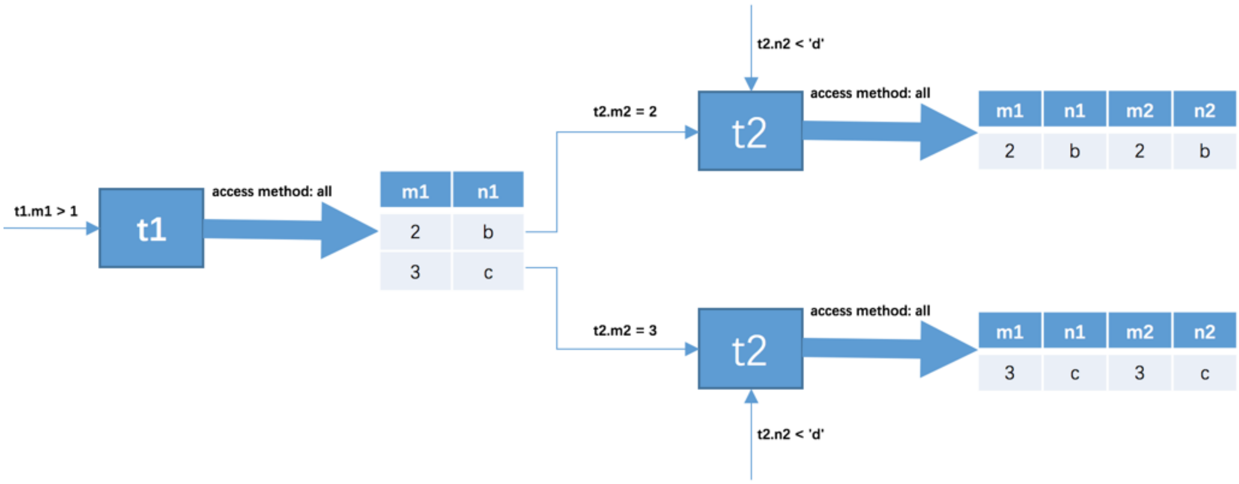
1. 首先确定第一个需要查询的表，这个表称之为 `驱动表` 。怎样在单表中执行查询语句我们在前一章都唠叨过了，只需要选取代价最小的那种访问方法去执行单表查询语句就好了（就是说从const、ref、ref_or_null、range、index、all这些执行方法中选取代价最小的去执行查询）。此处假设使用 `t1` 作为驱动表，那么就需要到 `t1` 表中找满足 `t1.m1 > 1` 的记录，因为表中的数据太少，我们也没在表上建立二级索引，所以此处查询 `t1` 表的访问方法就设定为 `all` 吧，也就是采用全表扫描的方式执行单表查询。关于如何提升连接查询的性能我们之后再说，现在先把基本概念捋清楚哈。所以查询过程就如下图所示：



   我们可以看到， `t1` 表中符合 `t1.m1 > 1` 的记录有两条。
2. 针对上一步骤中从驱动表产生的结果集中的每一条记录，分别需要到 `t2` 表中查找匹配的记录，所谓 `匹配的记录` ，指的是符合过滤条件的记录。因为是根据 `t1` 表中的记录去找 `t2` 表中的记录，所以 `t2` 表也可以被称之为 `被驱动表` 。上一步骤从驱动表中得到了2条记录，所以需要查询2次 `t2` 表。此时涉及两个表的列的过滤条件 `t1.m1 = t2.m2` 就派上用场了：

   - 当 `t1.m1 = 2` 时，过滤条件 `t1.m1 = t2.m2` 就相当于 `t2.m2 = 2` ，所以此时 `t2` 表相当于有了 `t2.m2 = 2` 、 `t2.n2 < 'd'` 这两个过滤条件，然后到 `t2` 表中执行单表查询。
   - 当 `t1.m1 = 3` 时，过滤条件 `t1.m1 = t2.m2` 就相当于 `t2.m2 = 3` ，所以此时 `t2` 表相当于有了 `t2.m2 = 3` 、 `t2.n2 < 'd'` 这两个过滤条件，然后到 `t2` 表中执行单表查询。

   所以整个连接查询的执行过程就如下图所示：



也就是说整个连接查询最后的结果只有两条符合过滤条件的记录：

```
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
```

从上边两个步骤可以看出来，我们上边唠叨的这个两表连接查询共需要查询1次 t1 表，2次 t2 表。当然这是在特定的过滤条件下的结果，如果我们把 t1.m1 > 1 这个条件去掉，那么从 t1 表中查出的记录就有3条，就需要查询3次 t2 表了。也就是说在两表连接查询中，驱动表只需要访问一次，被驱动表可能被访问多次。

11.1.3 内连接和外连接

为了大家更好理解后边内容，我们先创建两个有现实意义的表，

```
CREATE TABLE student (
    number INT NOT NULL AUTO_INCREMENT COMMENT '学号',
    name VARCHAR(5) COMMENT '姓名',
    major VARCHAR(30) COMMENT '专业',
    PRIMARY KEY (number)
) Engine=InnoDB CHARSET=utf8 COMMENT '学生信息表';

CREATE TABLE score (
    number INT COMMENT '学号',
    subject VARCHAR(30) COMMENT '科目',
    score TINYINT COMMENT '成绩',
    PRIMARY KEY (number, score)
) Engine=InnoDB CHARSET=utf8 COMMENT '学生成绩表';
```

我们新建了一个学生信息表，一个学生成绩表，然后我们向上述两个表中插入一些数据，为节省篇幅，具体插入过程就不唠叨了，插入后两表中的数据如下：

```
mysql> SELECT * FROM student;
+----------+-----------+--------------------------+
| number   | name      | major                    |
+----------+-----------+--------------------------+
| 20180101 | 杜子腾    | 软件学院                 |
| 20180102 | 范统      | 计算机科学与工程         |
| 20180103 | 史珍香    | 计算机科学与工程         |
+----------+-----------+--------------------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM score;
+----------+-----------------------------+-------+
| number   | subject                     | score |
+----------+-----------------------------+-------+
| 20180101 | 母猪的产后护理              |    78 |
| 20180101 | 论萨达姆的战争准备          |    88 |
| 20180102 | 论萨达姆的战争准备          |    98 |
| 20180102 | 母猪的产后护理              |   100 |
+----------+-----------------------------+-------+
4 rows in set (0.00 sec)
```

现在我们想把每个学生的考试成绩都查询出来就需要进行两表连接了（因为 `score` 中没有姓名信息，所以不能单纯只查询 `score` 表）。连接过程就是从 `student` 表中取出记录，在 `score` 表中查找 `number` 相同的成绩记录，所以过滤条件就是 `student.number = socre.number` ，整个查询语句就是这样：

```
mysql> SELECT * FROM student, score WHERE student.number = score.number;
+----------+-----------+--------------------------+----------+-----------------------------
-+-------+
| number   | name      | major                    | number   | subject
| score |
+----------+-----------+--------------------------+----------+-----------------------------
-+-------+
| 20180101 | 杜子腾    | 软件学院                 | 20180101 | 母猪的产后护理
|    78 |
| 20180101 | 杜子腾    | 软件学院                 | 20180101 | 论萨达姆的战争准备
|    88 |
| 20180102 | 范统      | 计算机科学与工程         | 20180102 | 论萨达姆的战争准备
|    98 |
| 20180102 | 范统      | 计算机科学与工程         | 20180102 | 母猪的产后护理
|   100 |
+----------+-----------+--------------------------+----------+-----------------------------
-+-------+
4 rows in set (0.00 sec)
```

字段有点多哦，我们少查询几个字段：

```
mysql> SELECT s1.number, s1.name, s2.subject, s2.score FROM student AS s1, score AS s2 WHE
RE s1.number = s2.number;
+----------+-----------+-----------------------------+-------+
| number   | name      | subject                     | score |
+----------+-----------+-----------------------------+-------+
| 20180101 | 杜子腾    | 母猪的产后护理              |    78 |
| 20180101 | 杜子腾    | 论萨达姆的战争准备          |    88 |
| 20180102 | 范统      | 论萨达姆的战争准备          |    98 |
| 20180102 | 范统      | 母猪的产后护理              |   100 |
+----------+-----------+-----------------------------+-------+
4 rows in set (0.00 sec)
```

从上述查询结果中我们可以看到，各个同学对应的各科成绩就都被查出来了，可是有个问题， 史珍香 同学，也就是学号为 20180103 的同学因为某些原因没有参加考试，所以在 score 表中没有对应的成绩记录。那如果老师想查看所有同学的考试成绩，即使是缺考的同学也应该展示出来，但是到目前为止我们介绍的 连接查询 是无法完成这样的需求的。我们稍微思考一下这个需求，其本质是想：驱动表中的记录即使在被驱动表中没有匹配的记录，也仍然需要加入到结果集。为了解决这个问题，就有了 内连接 和 外连接 的概念：

- 对于 内连接 的两个表，驱动表中的记录在被驱动表中找不到匹配的记录，该记录不会加入到最后的结果集，我们上边提到的连接都是所谓的 内连接 。
- 对于 外连接 的两个表，驱动表中的记录即使在被驱动表中没有匹配的记录，也仍然需要加入到结果集。

  在 MySQL 中，根据选取驱动表的不同，外连接仍然可以细分为2种：
  - 左外连接

    选取左侧的表为驱动表。
  - 右外连接

    选取右侧的表为驱动表。

可是这样仍然存在问题，即使对于外连接来说，有时候我们也并不想把驱动表的全部记录都加入到最后的结果集。这就犯难了，有时候匹配失败要加入结果集，有时候又不要加入结果集，这咋办，有点儿愁啊。。。噫，把过滤条件分为两种不就解决了这个问题了么，所以放在不同地方的过滤条件是有不同语义的：

- WHERE 子句中的过滤条件

  WHERE 子句中的过滤条件就是我们平时见的那种，不论是内连接还是外连接，凡是不符合 WHERE 子句中的过滤条件的记录都不会被加入最后的结果集。
- ON 子句中的过滤条件

  对于外连接的驱动表的记录来说，如果无法在被驱动表中找到匹配 ON 子句中的过滤条件的记录，那么该记录仍然会被加入到结果集中，对应的被驱动表记录的各个字段使用 NULL 值填充。

  需要注意的是，这个 ON 子句是专门为外连接驱动表中的记录在被驱动表找不到匹配记录时应不应该把该记录加入结果集这个场景下提出的，所以如果把 ON 子句放到内连接中， MySQL 会把它和 WHERE 子句一样对待，也就是说：内连接中的WHERE子句和ON子句是等价的。

一般情况下，我们都把只涉及单表的过滤条件放到 WHERE 子句中，把涉及两表的过滤条件都放到 ON 子句中，我们也一般把放到 ON 子句中的过滤条件也称之为 连接条件 。

```
小贴士：
左外连接和右外连接简称左连接和右连接，所以下边提到的左外连接和右外连接中的`外`字都用括号扩
起来，以表示这个字儿可有可无。
```

11.1.3.1 左（外）连接的语法

左（外）连接的语法还是挺简单的，比如我们要把 t1 表和 t2 表进行左外连接查询可以这么写：

```
SELECT * FROM t1 LEFT [OUTER] JOIN t2 ON 连接条件 [WHERE 普通过滤条件];
```

其中中括号里的 OUTER 单词是可以省略的。对于 LEFT JOIN 类型的连接来说，我们把放在左边的表称之为外表或者驱动表，右边的表称之为内表或者被驱动表。所以上述例子中 t1 就是外表或者驱动表， t2 就是内表或者被驱动表。需要注意的是，对于左（外）连接和右（外）连接来说，必须使用 ON 子句来指出连接条件。了解了左（外）连接的基本语法之后，再次回到我们上边那个现实问题中来，看看怎样写查询语句才能把所有的学生的成绩信息都查询出来，即使是缺考的考生也应该被放到结果集中：

```
mysql> SELECT s1.number, s1.name, s2.subject, s2.score FROM student AS s1 LEFT JOIN score
 AS s2 ON s1.number = s2.number;
+----------+-----------+-----------------------------+-------+
| number   | name      | subject                     | score |
+----------+-----------+-----------------------------+-------+
| 20180101 | 杜子腾    | 母猪的产后护理              |    78 |
| 20180101 | 杜子腾    | 论萨达姆的战争准备          |    88 |
| 20180102 | 范统      | 论萨达姆的战争准备          |    98 |
| 20180102 | 范统      | 母猪的产后护理              |   100 |
| 20180103 | 史珍香    | NULL                        |  NULL |
+----------+-----------+-----------------------------+-------+
5 rows in set (0.04 sec)
```

从结果集中可以看出来，虽然 史珍香 并没有对应的成绩记录，但是由于采用的是连接类型为左（外）连接，所以仍然把她放到了结果集中，只不过在对应的成绩记录的各列使用 NULL 值填充而已。

11.1.3.2 右（外）连接的语法

右（外）连接和左（外）连接的原理是一样一样的，语法也只是把 LEFT 换成 RIGHT 而已：

```
SELECT * FROM t1 RIGHT [OUTER] JOIN t2 ON 连接条件 [WHERE 普通过滤条件];
```

只不过驱动表是右边的表，被驱动表是左边的表，具体就不唠叨了。

11.1.3.3 内连接的语法

内连接和外连接的根本区别就是在驱动表中的记录不符合 ON 子句中的连接条件时不会把该记录加入到最后的结果集，我们最开始唠叨的那些连接查询的类型都是内连接。不过之前仅仅提到了一种最简单的内连接语法，就是直接把需要连接的多个表都放到 FROM 子句后边。其实针对内连接，MySQL提供了好多不同的语法，我们以 t1 和 t2 表为例瞅瞅：

```
SELECT * FROM t1 [INNER | CROSS] JOIN t2 [ON 连接条件] [WHERE 普通过滤条件];
```

也就是说在 MySQL 中，下边这几种内连接的写法都是等价的：

- SELECT * FROM t1 JOIN t2;
- SELECT * FROM t1 INNER JOIN t2;
- SELECT * FROM t1 CROSS JOIN t2;

上边的这些写法和直接把需要连接的表名放到 FROM 语句之后，用逗号 , 分隔开的写法是等价的：

```
SELECT * FROM t1, t2;
```

现在我们虽然介绍了很多种 内连接 的书写方式，不过熟悉一种就好了，这里我们推荐 INNER JOIN 的形式书写内连接（因为 INNER JOIN 语义很明确嘛，可以和 LEFT JOIN 和 RIGHT JOIN 很轻松的区分开）。这里需要注意的是，由于在内连接中ON子句和WHERE子句是等价的，所以内连接中不要求强制写明ON子句。

我们前边说过，连接的本质就是把各个连接表中的记录都取出来依次匹配的组合加入结果集并返回给用户。不论哪个表作为驱动表，两表连接产生的笛卡尔积肯定是一样的。而对于内连接来说，由于凡是不符合 ON 子句或 WHERE 子句中的条件的记录都会被过滤掉，其实也就相当于从两表连接的笛卡尔积中把不符合过滤条件的记录给踢出去，所以对于内连接来说，驱动表和被驱动表是可以互换的，并不会影响最后的查询结果。但是对于外连接来说，由于驱动表中的记录即使在被驱动表中找不到符合 ON 子句连接条件的记录，所以此时驱动表和被驱动表的关系就很重要了，也就是说左外连接和右外连接的驱动表和被驱动表不能轻易互换。

11.1.3.4 小结

上边说了很多，给大家的感觉不是很直观，我们直接把表 t1 和 t2 的三种连接方式写在一起，这样大家理解起来就很easy了：

```
mysql> SELECT * FROM t1 INNER JOIN t2 ON t1.m1 = t2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
|    1 | a    | NULL | NULL |
+------+------+------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM t1 RIGHT JOIN t2 ON t1.m1 = t2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
| NULL | NULL |    4 | d    |
+------+------+------+------+
3 rows in set (0.00 sec)
```

11.2 连接的原理

上边贼啰嗦的介绍都只是为了唤醒大家对 连接 、 内连接 、 外连接 这些概念的记忆，这些基本概念是为了真正进入本章主题做的铺垫。真正的重点是MySQL采用了什么样的算法来进行表与表之间的连接，了解了这个之后，大家才能明白为啥有的连接查询运行的快如闪电，有的却慢如蜗牛。

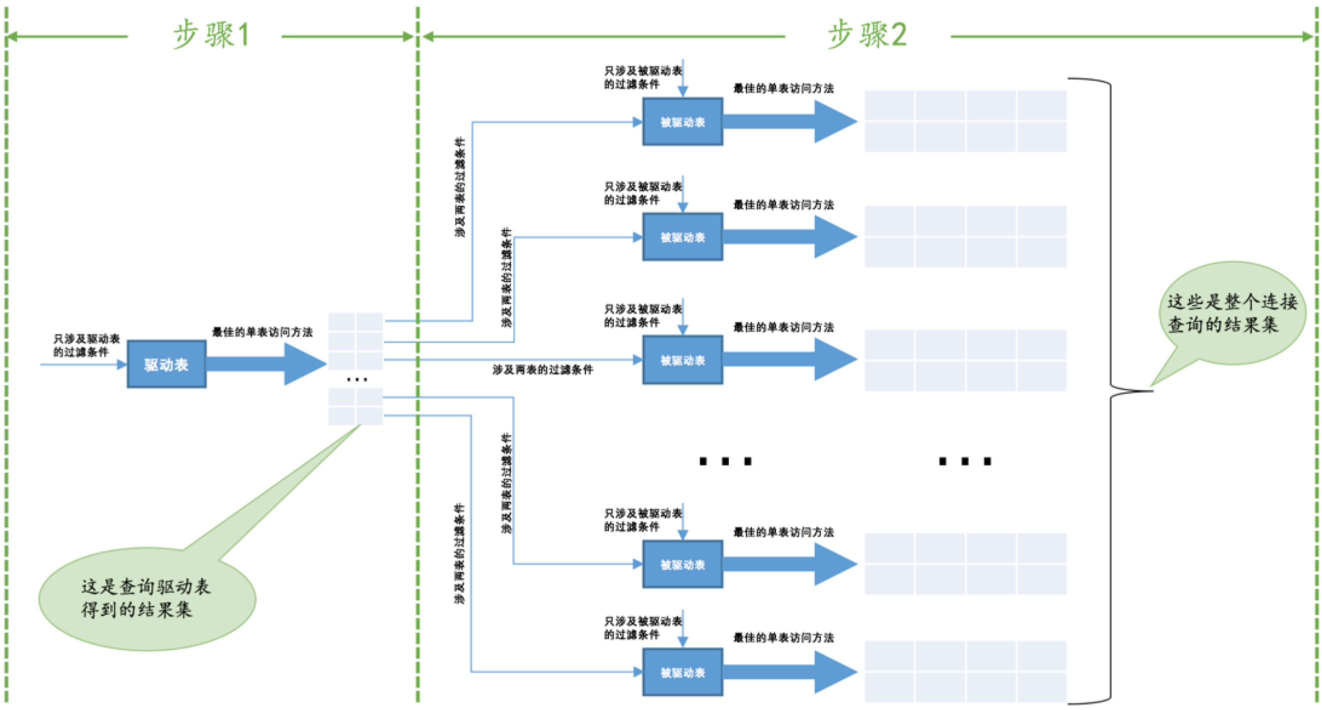
11.2.1 嵌套循环连接（Nested-Loop Join）

我们前边说过，对于两表连接来说，驱动表只会被访问一遍，但被驱动表却要被访问到好多遍，具体访问几遍取决于对驱动表执行单表查询后的结果集中的记录条数。对于内连接来说，选取哪个表为驱动表都没关系，而外连接的驱动表是固定的，也就是说左（外）连接的驱动表就是左边的那个表，右（外）连接的驱动表就是右边的那

个表。我们上边已经大致介绍过 t1 表和 t2 表执行内连接查询的大致过程，我们温习一下：

- 步骤1：选取驱动表，使用与驱动表相关的过滤条件，选取代价最低的单表访问方法来执行对驱动表的单表查询。
- 步骤2：对上一步骤中查询驱动表得到的结果集中每一条记录，都分别到被驱动表中查找匹配的记录。

通用的两表连接过程如下图所示：



如果有3个表进行连接的话，那么 步骤2 中得到的结果集就像是新的驱动表，然后第三个表就成为了被驱动表，重复上边过程，也就是 步骤2 中得到的结果集中的每一条记录都需要到 t3 表中找一找有没有匹配的记录，用伪代码表示一下这个过程就是这样：

```
for each row in t1 {   #此处表示遍历满足对t1单表查询结果集中的每一条记录

    for each row in t2 {   #此处表示对于某条t1表的记录来说，遍历满足对t2单表查询结果集中的每一条记录

        for each row in t3 {   #此处表示对于某条t1和t2表的记录组合来说，对t3表进行单表查询
            if row satisfies join conditions, send to client
        }
    }
}
```

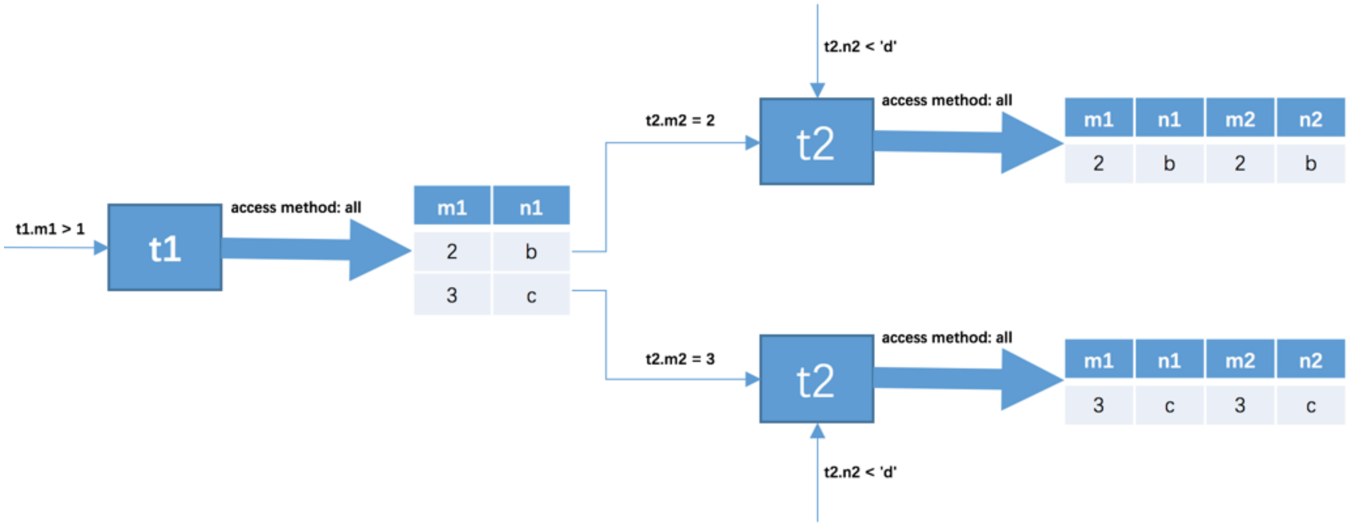
这个过程就像是一个嵌套的循环，所以这种驱动表只访问一次，但被驱动表却可能被多次访问，访问次数取决于对驱动表执行单表查询后的结果集中的记录条数的连接执行方式称之为 嵌套循环连接 （ Nested-Loop Join ），这是最简单，也是最笨拙的一种连接查询算法。

11.2.2 使用索引加快连接速度

我们知道在 嵌套循环连接 的 步骤2 中可能需要访问多次被驱动表，如果访问被驱动表的方式都是全表扫描的话，妈呀，那得要扫描好多次呀~ ~ ~ 但是别忘了，查询 t2 表其实就相当于一次单表扫描，我们可以利用索引来加快查询速度哦。回顾一下最开始介绍的 t1 表和 t2 表进行内连接的例子：

```
SELECT * FROM t1, t2 WHERE t1.m1 > 1 AND t1.m1 = t2.m2 AND t2.n2 < 'd';
```

我们使用的其实是 嵌套循环连接 算法执行的连接查询，再把上边那个查询执行过程表拉下来给大家看一下：



查询驱动表 t1 后的结果集中有两条记录， 嵌套循环连接 算法需要对被驱动表查询2次：

- 当 t1.m1 = 2 时，去查询一遍 t2 表，对 t2 表的查询语句相当于：

```
SELECT * FROM t2 WHERE t2.m2 = 2 AND t2.n2 < 'd';
```

- 当 t1.m1 = 3 时，再去查询一遍 t2 表，此时对 t2 表的查询语句相当于：

```
SELECT * FROM t2 WHERE t2.m2 = 3 AND t2.n2 < 'd';
```

可以看到，原来的 t1.m1 = t2.m2 这个涉及两个表的过滤条件在针对 t2 表做查询时关于 t1 表的条件就已经确定了，所以我们只需要单单优化对 t2 表的查询了，上述两个对 t2 表的查询语句中利用到的列是 m2 和 n2 列，我们可以：

- 在 m2 列上建立索引，因为对 m2 列的条件是等值查找，比如 t2.m2 = 2 、 t2.m2 = 3 等，所以可能使用到 ref 的访问方法，假设使用 ref 的访问方法去执行对 t2 表的查询的话，需要回表之后再判断 t2.n2 < d 这个条件是否成立。

  这里有一个比较特殊的情况，就是假设 m2 列是 t2 表的主键或者唯一二级索引列，那么使用 t2.m2 = 常数值 这样的条件从 t2 表中查找记录的过程的代价就是常数级别的。我们知道在单表中使用主键值或者唯一二级索引列的值进行等值查找的方式称之为 const ，而设计 MySQL 的大叔把在连接查询中对被驱动表使用主键值或者唯一二级索引列的值进行等值查找的查询执行方式称之为： eq_ref 。
- 在 n2 列上建立索引，涉及到的条件是 t2.n2 < 'd' ，可能用到 range 的访问方法，假设使用 range 的访问方法对 t2 表的查询的话，需要回表之后再判断在 m2 列上的条件是否成立。

假设 m2 和 n2 列上都存在索引的话，那么就需要从这两个里边儿挑一个代价更低的去执行对 t2 表的查询。当然，建立了索引不一定使用索引，只有在 二级索引 + 回表 的代价比全表扫描的代价更低时才会使用索引。

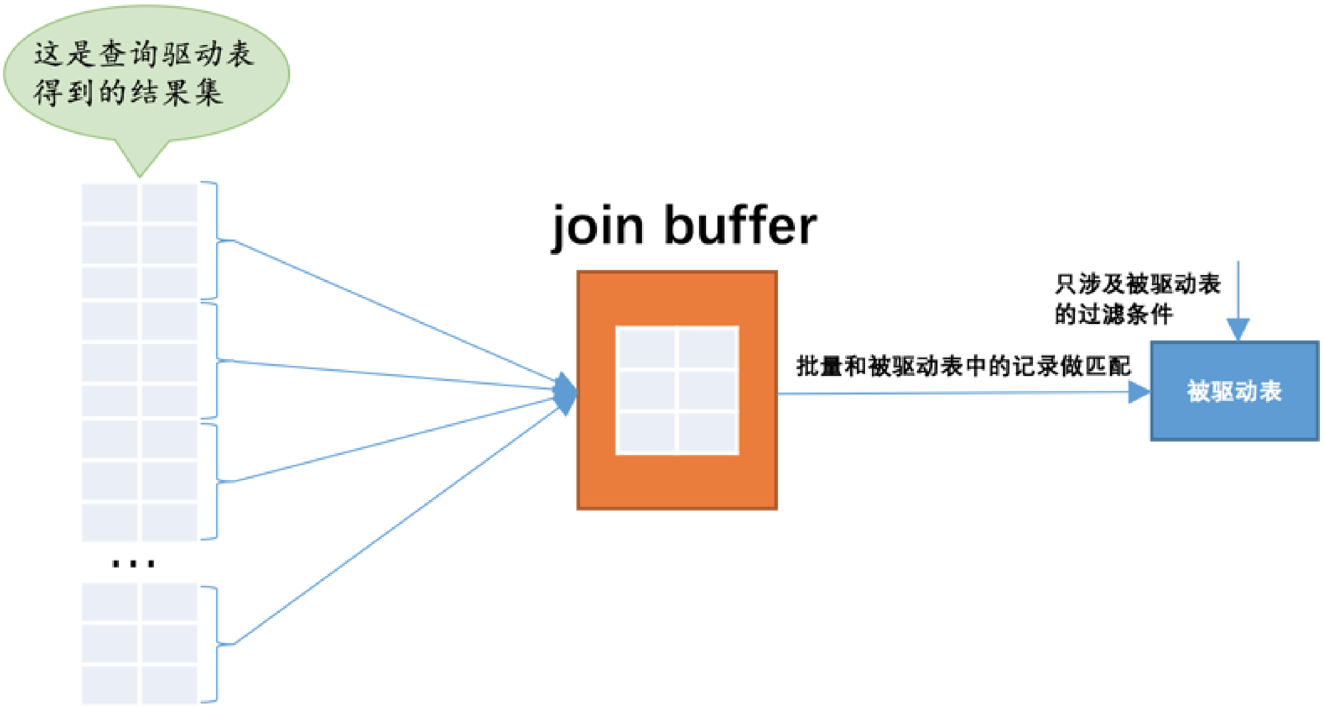
另外，有时候连接查询的查询列表和过滤条件中可能只涉及被驱动表的部分列，而这些列都是某个索引的一部分，这种情况下即使不能使用 eq_ref 、 ref 、 ref_or_null 或者 range 这些访问方法执行对被驱动表的查询的话，也可以使用索引扫描，也就是 index 的访问方法来查询被驱动表。所以我们建议在真实工作中最好不要使用 * 作为查询列表，最好把真实用到的列作为查询列表。

11.2.3 基于块的嵌套循环连接（Block Nested-Loop Join）

扫描一个表的过程其实是先把这个表从磁盘上加载到内存中，然后从内存中比较匹配条件是否满足。现实生活中的表可不像 t1 、 t2 这种只有3条记录，成千上万条记录都是少的，几百万、几千万甚至几亿条记录的表到处都是。内存里可能并不能完全存放的下表中所有的记录，所以在扫描表前边记录的时候后边的记录可能还在磁盘

上，等扫描到后边记录的时候可能内存不足，所以需要把前边的记录从内存中释放掉。我们前边又说过，采用 嵌套循环连接 算法的两表连接过程中，被驱动表可是要被访问好多次的，如果这个被驱动表中的数据特别多而且不能使用索引进行访问，那就相当于要从磁盘上读好几次这个表，这个 I/O 代价就非常大了，所以我们得想办法：尽量减少访问被驱动表的次数。

当被驱动表中的数据非常多时，每次访问被驱动表，被驱动表的记录会被加载到内存中，在内存中的每一条记录只会和驱动表结果集的一条记录做匹配，之后就会被从内存中清除掉。然后再从驱动表结果集中拿出另一条记录，再一次把被驱动表的记录加载到内存中一遍，周而复始，驱动表结果集中有多少条记录，就得把被驱动表从磁盘上加载到内存中多少次。所以我们可不可以在把被驱动表的记录加载到内存的时候，一次性和多条驱动表中的记录做匹配，这样就可以大大减少重复从磁盘上加载被驱动表的代价了。所以设计 MySQL 的大叔提出了一个 join buffer 的概念， join buffer 就是执行连接查询前申请的一块固定大小的内存，先把若干条驱动表结果集中的记录装在这个 join buffer 中，然后开始扫描被驱动表，每一条被驱动表的记录一次性和 join buffer 中的多条驱动表记录做匹配，因为匹配的过程都是在内存中完成的，所以这样可以显著减少被驱动表的 I/O 代价。使用 join buffer 的过程如下图所示：



最好的情况是 join buffer 足够大，能容纳驱动表结果集中的所有记录，这样只需要访问一次被驱动表就可以完成连接操作了。设计 MySQL 的大叔把这种加入了 join buffer 的嵌套循环连接算法称之为 基于块的嵌套连接 （Block Nested-Loop Join）算法。

这个 join buffer 的大小是可以通过启动参数或者系统变量 join_buffer_size 进行配置，默认大小为 262144字节 （也就是 256KB ），最小可以设置为 128字节 。当然，对于优化被驱动表的查询来说，最好是为被驱动表加上效率高的索引，如果实在不能使用索引，并且自己的机器的内存也比较大可以尝试调大 join_buffer_size 的值来对连接查询进行优化。

另外需要注意的是，驱动表的记录并不是所有列都会被放到 join buffer 中，只有查询列表中的列和过滤条件中的列才会被放到 join buffer 中，所以再次提醒我们，最好不要把 * 作为查询列表，只需要把我们关心的列放到查询列表就好了，这样还可以在 join buffer 中放置更多的记录呢哈。

12 第12章 谁最便宜就选谁-MySQL基于成本的优化

标签： MySQL是怎样运行的

12.1 什么是成本

我们之前老说 MySQL 执行一个查询可以有不同的执行方案，它会选择其中成本最低，或者说代价最低的那种方案去真正的执行查询。不过我们之前对 成本 的描述是非常模糊的，其实在 MySQL 中一条查询语句的执行成本是由下边这两个方面组成的：

- I/O 成本

  我们的表经常使用的 MyISAM 、 InnoDB 存储引擎都是将数据和索引都存储到磁盘上的，当我们想查询表中的记录时，需要先把数据或者索引加载到内存中然后再操作。这个从磁盘到内存这个加载的过程损耗的时间称之为 I/O 成本。
- CPU 成本

  读取以及检测记录是否满足对应的搜索条件、对结果集进行排序等这些操作损耗的时间称之为 CPU 成本。

对于 InnoDB 存储引擎来说，页是磁盘和内存之间交互的基本单位，设计 MySQL 的大叔规定读取一个页面花费的成本默认是 1.0 ，读取以及检测一条记录是否符合搜索条件的成本默认是 0.2 。 1.0 、 0.2 这些数字称之为 成本常数 ，这两个成本常数我们最常用到，其余的成本常数我们后边再说哈。

> 小贴士：
> 需要注意的是，不管读取记录时需不需要检测是否满足搜索条件，其成本都算是0.2。

12.2 单表查询的成本

12.2.1 准备工作

为了故事的顺利发展，我们还得把之前用到的 single_table 表搬来，怕大家忘了这个表长啥样，再给大家抄一遍：

```
CREATE TABLE single_table (
    id INT NOT NULL AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1),
    UNIQUE KEY idx_key2 (key2),
    KEY idx_key3 (key3),
    KEY idx_key_part(key_part1, key_part2, key_part3)
) Engine=InnoDB CHARSET=utf8;
```

还是假设这个表里边儿有10000条记录，除 id 列外其余的列都插入随机值。下边正式开始我们的表演。

12.2.2 基于成本的优化步骤

在一条单表查询语句真正执行之前， MySQL 的查询优化器会找出执行该语句所有可能使用的方案，对比之后找出成本最低的方案，这个成本最低的方案就是所谓的 执行计划 ，之后才会调用存储引擎提供的接口真正的执行查询，这个过程总结一下就是这样：

1. 根据搜索条件，找出所有可能使用的索引
2. 计算全表扫描的代价
3. 计算使用不同索引执行查询的代价
4. 对比各种执行方案的代价，找出成本最低的那一个

下边我们就以一个实例来分析一下这些步骤，单表查询语句如下：

```
SELECT * FROM single_table WHERE
    key1 IN ('a', 'b', 'c') AND
    key2 > 10 AND key2 < 1000 AND
    key3 > key2 AND
    key_part1 LIKE '%hello%' AND
    common_field = '123';
```

乍看上去有点儿复杂哦，我们一步一步分析一下。

12.2.2.1 1. 根据搜索条件，找出所有可能使用的索引

我们前边说过，对于 B+ 树索引来说，只要索引列和常数使用 = 、 <=> 、 IN 、 NOT IN 、 IS NULL 、 IS NOT NULL 、 > 、 < 、 >= 、 <= 、 BETWEEN 、 != （不等于也可以写成 <> ）或者 LIKE 操作符连接起来，就可以产生一个所谓的 范围区间 （ LIKE 匹配字符串前缀也行），也就是说这些搜索条件都可能使用到索引，设计 MySQL 的大叔把一个查询中可能使用到的索引称之为 possible keys 。

我们分析一下上边查询中涉及到的几个搜索条件：

- key1 IN ('a', 'b', 'c') ，这个搜索条件可以使用二级索引 idx_key1 。
- key2 > 10 AND key2 < 1000 ，这个搜索条件可以使用二级索引 idx_key2 。
- key3 > key2 ，这个搜索条件的索引列由于没有和常数比较，所以并不能使用到索引。
- key_part1 LIKE '%hello%' ， key_part1 通过 LIKE 操作符和以通配符开头的字符串做比较，不可以适用索引。
- common_field = '123' ，由于该列上压根儿没有索引，所以不会用到索引。

综上所述，上边的查询语句可能用到的索引，也就是 possible keys 只有 idx_key1 和 idx_key2 。

12.2.2.2 2. 计算全表扫描的代价

对于 InnoDB 存储引擎来说，全表扫描的意思就是把聚簇索引中的记录都依次和给定的搜索条件做一下比较，把符合搜索条件的记录加入到结果集，所以需要将聚簇索引对应的页面加载到内存中，然后再检测记录是否符合搜索条件。由于查询成本= I/O 成本+ CPU 成本，所以计算全表扫描的代价需要两个信息：

- 聚簇索引占用的页面数
- 该表中的记录数

这两个信息从哪来呢？设计 MySQL 的大叔为每个表维护了一系列的 统计信息 ，关于这些统计信息是如何收集起来的我们放在本章后边详细唠叨，现在看看怎么查看这些统计信息哈。设计 MySQL 的大叔给我们提供了 SHOW TABLE STATUS 语句来查看表的统计信息，如果要看指定的某个表的统计信息，在该语句后加对应的 LIKE 语句就好了，比方说我们要查看 single_table 这个表的统计信息可以这么写：

```
mysql> USE xiaohaizi;
Database changed

mysql> SHOW TABLE STATUS LIKE 'single_table'\G
*************************** 1. row ***************************
           Name: single_table
         Engine: InnoDB
        Version: 10
     Row_format: Dynamic
           Rows: 9693
 Avg_row_length: 163
    Data_length: 1589248
Max_data_length: 0
   Index_length: 2752512
      Data_free: 4194304
 Auto_increment: 10001
    Create_time: 2018-12-10 13:37:23
    Update_time: 2018-12-10 13:38:03
     Check_time: NULL
      Collation: utf8_general_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.01 sec)
```

虽然出现了很多统计选项，但我们目前只关心两个：

- Rows

  本选项表示表中的记录条数。对于使用 MyISAM 存储引擎的表来说，该值是准确的，对于使用 InnoDB 存储引擎的表来说，该值是一个估计值。从查询结果我们也可以看出来，由于我们的 single_table 表是使用 InnoDB 存储引擎的，所以虽然实际上表中有10000条记录，但是 SHOW TABLE STATUS 显示的 Rows 值只有9693条记录。

- Data_length

  本选项表示表占用的存储空间字节数。使用 MyISAM 存储引擎的表来说，该值就是数据文件的大小，对于使用 InnoDB 存储引擎的表来说，该值就相当于聚簇索引占用的存储空间大小，也就是说可以这样计算该值的大小：

  ```
  Data_length = 聚簇索引的页面数量 x 每个页面的大小
  ```

  我们的 single_table 使用默认 16KB 的页面大小，而上边查询结果显示 Data_length 的值是 1589248 ，所以我们可以反向来推导出 聚簇索引的页面数量 ：

  ```
  聚簇索引的页面数量 = 1589248 ÷ 16 ÷ 1024 = 97
  ```

我们现在已经得到了聚簇索引占用的页面数量以及该表记录数的估计值，所以就可以计算全表扫描成本了，但是设计 MySQL 的大叔在真实计算成本时会进行一些 微调 ，这些微调的值是直接硬编码到代码里的，由于没有注释，我也不知道这些微调值是个啥子意思，但是由于这些微调的值十分的小，并不影响我们分析，所以我们也没有必要在这些微调值上纠结了。现在可以看一下全表扫描成本的计算过程：

- I/O 成本

  ```
  97 x 1.0 + 1.1 = 98.1
  ```

97 指的是聚簇索引占用的页面数， 1.0 指的是加载一个页面的成本常数，后边的 1.1 是一个微调值，我们不用在意。

- CPU 成本:

```
9693 x 0.2 + 1.0 = 1939.6
```

  9693 指的是统计数据中表的记录数，对于 InnoDB 存储引擎来说是一个估计值， 0.2 指的是访问一条记录所需的成本常数，后边的 1.0 是一个微调值，我们不用在意。

- 总成本:

```
98.1 + 1939.6 = 2037.7
```

综上所述，对于 single_table 的全表扫描所需的总成本就是 2037.7 。

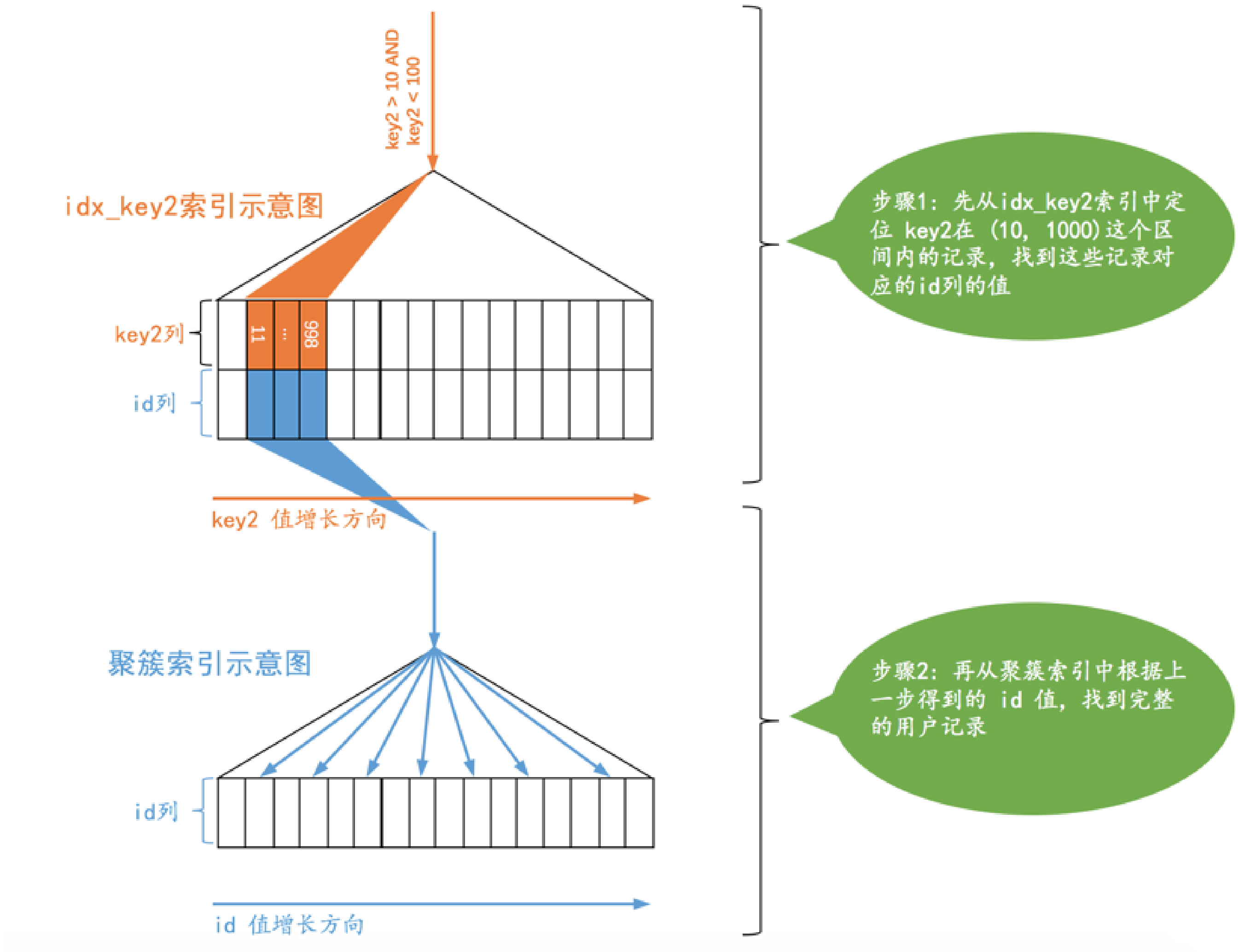
> 小贴士：
> 我们前边说过表中的记录其实都存储在聚簇索引对应B+树的叶子节点中，所以只要我们通过根节点获得了最左边的叶子节点，就可以沿着叶子节点组成的双向链表把所有记录都查看一遍。也就是说全表扫描这个过程其实有的B+树内节点是不需要访问的，但是设计MySQL的大叔们在计算全表扫描成本时直接使用聚簇索引占用的页面数作为计算I/O成本的依据，是不区分内节点和叶子节点的，有点儿简单暴力，大家注意一下就好了。

12.2.2.3 3. 计算使用不同索引执行查询的代价

从第1步分析我们得到，上述查询可能使用到 idx_key1 和 idx_key2 这两个索引，我们需要分别分析单独使用这些索引执行查询的成本，最后还要分析是否可能使用到索引合并。这里需要提一点的是， MySQL 查询优化器先分析使用唯一二级索引的成本，再分析使用普通索引的成本，所以我们也先分析 idx_key2 的成本，然后再看使用 idx_key1 的成本。

使用idx_key2执行查询的成本分析

idx_key2 对应的搜索条件是： key2 > 10 AND key2 < 1000 ，也就是说对应的范围区间就是： (10, 1000) ，使用 idx_key2 搜索的示意图就是这样子:



对于使用 二级索引 + 回表 方式的查询，设计 MySQL 的大叔计算这种查询的成本依赖两个方面的数据：

- 范围区间数量

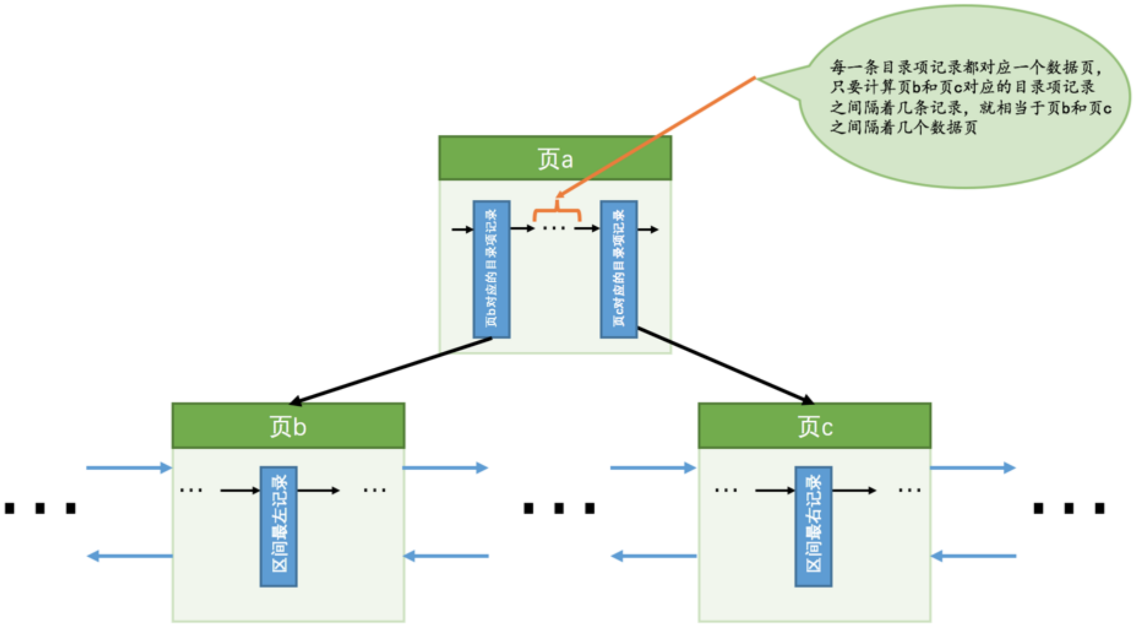
  不论某个范围区间的二级索引到底占用了多少页面，查询优化器粗暴的认为读取索引的一个范围区间的 I/O 成本和读取一个页面是相同的。本例中使用 idx_key2 的范围区间只有一个：（10, 1000），所以相当于访问这个范围区间的二级索引付出的 I/O 成本就是：

  ```
  1 x 1.0 = 1.0
  ```

- 需要回表的记录数

  优化器需要计算二级索引的某个范围区间到底包含多少条记录，对于本例来说就是要计算 idx_key2 在 (10, 1000) 这个范围区间中包含多少二级索引记录，计算过程是这样的：

  - 步骤1：先根据 key2 > 10 这个条件访问一下 idx_key2 对应的 B+ 树索引，找到满足 key2 > 10 这个条件的第一条记录，我们把这条记录称之为 区间最左记录 。我们前头说过在 B+ 数树中定位一条记录的过程是贼快的，是常数级别的，所以这个过程的性能消耗是可以忽略不计的。
    - 步骤2：然后再根据 key2 < 1000 这个条件继续从 idx_key2 对应的 B+ 树索引中找出第一条满足这个条件的记录，我们把这条记录称之为 区间最右记录 ，这个过程的性能消耗也可以忽略不计的。
    - 步骤3：如果 区间最左记录 和 区间最右记录 相隔不太远（在 MySQL 5.7.21 这个版本里，只要相隔不大于10个页面即可），那就可以精确统计出满足 key2 > 10 AND key2 < 1000 条件的二级索引记录条数。否则只沿着 区间最左记录 向右读10个页面，计算平均每个页面中包含多少记录，然后用这个平均值乘以 区间最左记录 和 区间最右记录 之间的页面数量就可以了。那么问题又来了，怎么估计 区间最左记录 和 区间最右记录 之间有多少个页面呢？解决这个问题还得回到 B+ 树索引的结构中来：



如图，我们假设 区间最左记录 在 页b 中， 区间最右记录 在 页c 中，那么我们想计算 区间最左记录 和 区间最右记录 之间的页面数量就相当于计算 页b 和 页c 之间有多少页面，而每一条 目录项记录 都对应一个数据页，所以计算 页b 和 页c 之间有多少页面就相当于计算它们父节点（也就是页a）中对应的目录项记录之间隔着几条记录。在一个页面中统计两条记录之间有几条记录的成本就贼小了。

不过还有问题，如果 页b 和 页c 之间的页面实在太多，以至于 页b 和 页c 对应的目录项记录都不在一个页面中该咋办？继续递归啊，也就是再统计 页b 和 页c 对应的目录项记录所在页之间有多少个页面。之前我们说过一个 B+ 树有4层高已经很了不得了，所以这个统计过程也不是很耗费性能。

知道了如何统计二级索引某个范围区间的记录数之后，就需要回到现实问题中来，根据上述算法测得 idx_key2 在区间（10， 1000）之间大约有 95 条记录。读取这 95 条二级索引记录需要付出的 CPU 成本就是：

```
95 x 0.2 + 0.01 = 19.01
```

其中 95 是需要读取的二级索引记录条数， 0.2 是读取一条记录成本常数， 0.01 是微调。

在通过二级索引获取到记录之后，还需要干两件事儿：

- 根据这些记录里的主键值到聚簇索引中做回表操作

  这里需要大家使劲儿睁大自己滴溜溜的大眼睛仔细瞧，设计 MySQL 的大叔评估回表操作的 I/O 成本依旧很豪放，他们认为每次回表操作都相当于访问一个页面，也就是说二级索引范围区间有多少记录，就需要进行多少次回表操作，也就是需要进行多少次页面 I/O 。我们上边统计了使用 idx_key2 二级索引执行查询时，预计有 95 条二级索引记录需要进行回表操作，所以回表操作带来的 I/O 成本就是：

  ```
  95 x 1.0 = 95.0
  ```

  其中 95 是预计的二级索引记录数， 1.0 是一个页面的 I/O 成本常数。

- 回表操作后得到的完整用户记录，然后再检测其他搜索条件是否成立

  回表操作的本质就是通过二级索引记录的主键值到聚簇索引中找到完整的用户记录，然后再检测除 key2 > 10 AND key2 < 1000 这个搜索条件以外的搜索条件是否成立。因为我们通过范围区间获取到二级索引记录共 95 条，也就对应着聚簇索引中 95 条完整的用户记录，读取并检测这些完整的用户记录是否符合其余的搜索条件的 CPU 成本如下：

设计`MySQL`的大叔只计算这个查找过程所需的`I/O`成本，也就是我们上一步骤中得到的`95.0`，在内存中的定位完整用户记录的过程的成本是忽略不计的。在定位到这些完整的用户记录后，需要检测除`key2 > 10 AND key2 < 1000`这个搜索条件以外的搜索条件是否成立，这个比较过程花费的`CPU`成本就是：

```
95 x 0.2 = 19.0
```

其中`95`是待检测记录的条数，`0.2`是检测一条记录是否符合给定的搜索条件的成本常数。

所以本例中使用 idx_key2 执行查询的成本就如下所示:

- I/O 成本:

  1.0 + 95 x 1.0 = 96.0 (范围区间的数量 + 预估的二级索引记录条数)

- CPU 成本:

  95 x 0.2 + 0.01 + 95 x 0.2 = 38.01 （读取二级索引记录的成本 + 读取并检测回表后聚簇索引记录的成本）

综上所述，使用 idx_key2 执行查询的总成本就是:

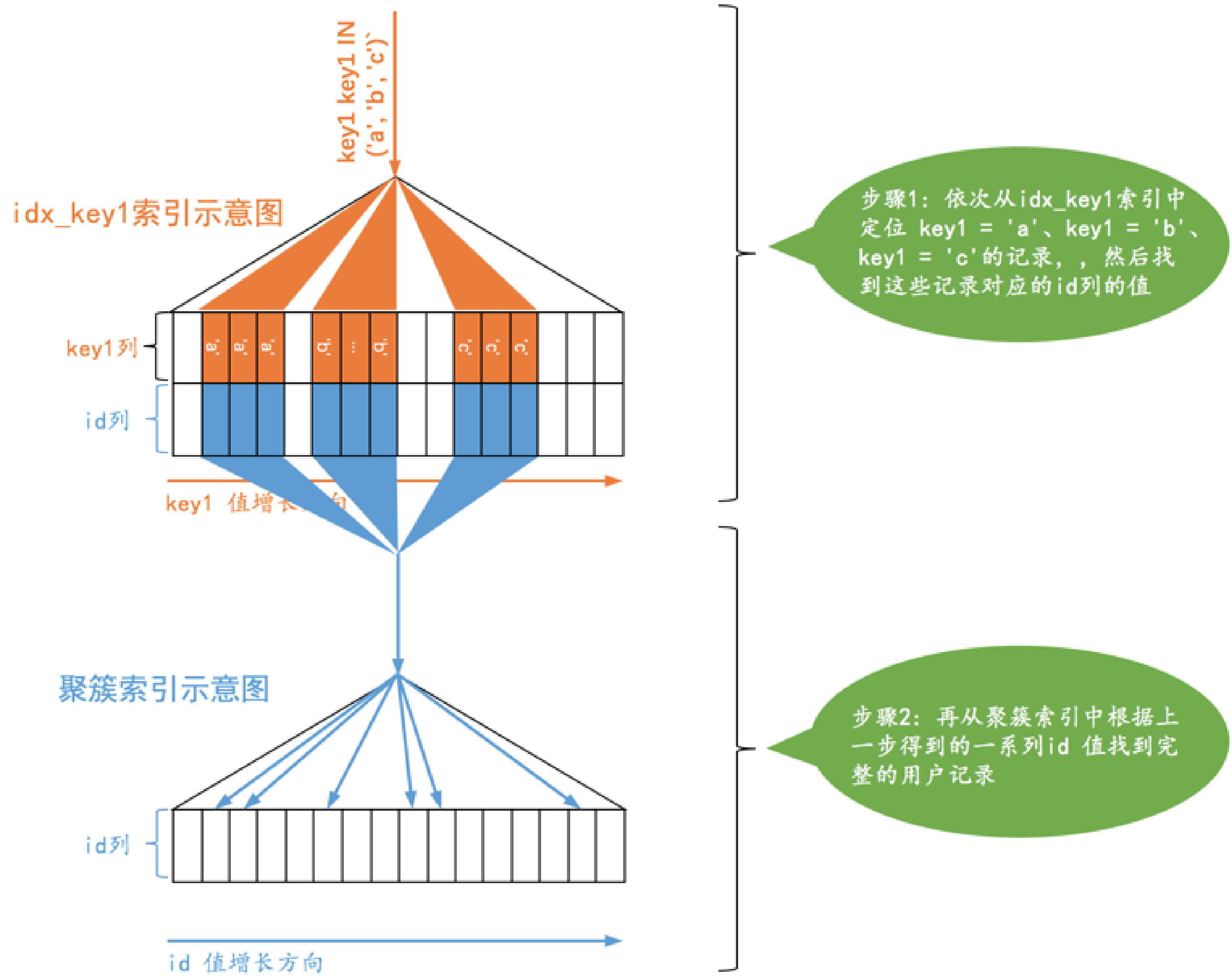
96.0 + 38.01 = 134.01

使用idx_key1执行查询的成本分析

idx_key1 对应的搜索条件是： key1 IN ('a', 'b', 'c')，也就是说相当于3个单点区间:

- ['a', 'a']
- ['b', 'b']
- ['c', 'c']

使用 idx_key1 搜索的示意图就是这样子:



与使用 idx_key2 的情况类似，我们也需要计算使用 idx_key1 时需要访问的范围区间数量以及需要回表的记录数：

- 范围区间数量

  使用 idx_key1 执行查询时很显然有3个单点区间，所以访问这3个范围区间的二级索引付出的I/O成本就是：

  ```
  3 x 1.0 = 3.0
  ```

- 需要回表的记录数

  由于使用 idx_key1 时有3个单点区间，所以每个单点区间都需要查找一遍对应的二级索引记录数：

  - 查找单点区间 ['a', 'a'] 对应的二级索引记录数

    计算单点区间对应的二级索引记录数和计算连续范围区间对应的二级索引记录数是一样的，都是先计算 区间最左记录 和 区间最右记录 ，然后再计算它们之间的记录数，具体算法上边都唠叨过了，就不赘述了。最后计算得到单点区间 ['a', 'a'] 对应的二级索引记录数是： 35 。

  - 查找单点区间 ['b', 'b'] 对应的二级索引记录数

    与上同理，计算得到本单点区间对应的记录数是： 44 。

  - 查找单点区间 ['c', 'c'] 对应的二级索引记录数

    与上同理，计算得到本单点区间对应的记录数是： 39 。

    所以，这三个单点区间总共需要回表的记录数就是：

    ```
    35 + 44 + 39 = 118
    ```

读取这些二级索引记录的 CPU 成本就是：

```
118 x 0.2 + 0.01 = 23.61
```

得到总共需要回表的记录数之后，就要考虑：

- 根据这些记录里的主键值到聚簇索引中做回表操作

  所需的 I/O 成本就是：

  ```
  118 x 1.0 = 118.0
  ```

- 回表操作后得到的完整用户记录，然后再比较其他搜索条件是否成立

  此步骤对应的 CPU 成本就是：

  ```
  118 x 0.2 = 23.6
  ```

所以本例中使用 idx_key1 执行查询的成本就如下所示：

- I/O 成本：

  ```
  3.0 + 118 x 1.0 = 121.0 (范围区间的数量 + 预估的二级索引记录条数)
  ```

- CPU 成本：

  ```
  118 x 0.2 + 0.01 + 118 x 0.2 = 47.21 （读取二级索引记录的成本 + 读取并检测回表后聚簇索引记录的成本）
  ```

综上所述，使用 idx_key1 执行查询的总成本就是：

```
121.0 + 47.21 = 168.21
```

是否有可能使用索引合并（Index Merge）

本例中有关 key1 和 key2 的搜索条件是使用 AND 连接起来的，而对于 idx_key1 和 idx_key2 都是范围查询，也就是说查找到的二级索引记录并不是按照主键值进行排序的，并不满足使用 Intersection 索引合并的条件，所以并不会使用索引合并。

> 小贴士：
> MySQL查询优化器计算索引合并成本的算法也比较麻烦，所以我们这也就不展开唠叨了。

12.2.2.4 4. 对比各种执行方案的代价，找出成本最低的那一个

下边把执行本例中的查询的各种可执行方案以及它们对应的成本列出来：

- 全表扫描的成本： 2037.7
- 使用 idx_key2 的成本： 134.01
- 使用 idx_key1 的成本： 168.21

很显然，使用 idx_key2 的成本最低，所以当然选择 idx_key2 来执行查询喽。

> 小贴士：
> 考虑到大家的阅读体验，为了最大限度的减少大家在理解优化器工作原理的过程中遇到的懵逼情况，这里对优化器在单表查询中对比各种执行方案的代价的方式稍稍的做了简化，不过毕竟大部分同学不需要去看MySQL的源码，把大致的精神传递正确就好了哈。

12.2.3 基于索引统计数据的成本计算

有时候使用索引执行查询时会有许多单点区间，比如使用 IN 语句就很容易产生非常多的单点区间，比如下边这个查询（下边查询语句中的 ... 表示还有很多参数）：

```
SELECT * FROM single_table WHERE key1 IN ('aa1', 'aa2', 'aa3', ... , 'zzz');
```

很显然，这个查询可能使用到的索引就是 idx_key1 ，由于这个索引并不是唯一二级索引，所以并不能确定一个单点区间对应的二级索引记录的条数有多少，需要我们去计算。计算方式我们上边已经介绍过了，就是先获取索引对应的 B+ 树的 区间最左记录 和 区间最右记录 ，然后再计算这两条记录之间有多少记录（记录条数少的时候可以做到精确计算，多的时候只能估算）。设计 MySQL 的大叔把这种通过直接访问索引对应的 B+ 树来计算某个范围区间对应的索引记录条数的方式称之为 index dive 。

> 小贴士：
> dive直译为中文的意思是跳水、俯冲的意思，原谅我的英文水平捉急，我实在不知道怎么翻译 index dive，索引跳水？索引俯冲？好像都不太合适，所以压根儿就不翻译了。不过大家要意会index dive就是直接利用索引对应的B+树来计算某个范围区间对应的记录条数。

有零星几个单点区间的话，使用 index dive 的方式去计算这些单点区间对应的记录数也不是什么问题，可是你架不住有的孩子憋足了劲往 IN 语句里塞东西呀，我就见过有的同学写的 IN 语句里有20000个参数的🤣🤣，这就意味着 MySQL 的查询优化器为了计算这些单点区间对应的索引记录条数，要进行20000次 index dive 操作，这性能损耗可就大了，搞不好计算这些单点区间对应的索引记录条数的成本比直接全表扫描的成本都大了。设计 MySQL 的大叔们多聪明啊，他们当然考虑到了这种情况，所以提供了一个系统变量 eq_range_index_dive_limit ，我们看一下在 MySQL 5.7.21 中这个系统变量的默认值：

```
mysql> SHOW VARIABLES LIKE '%dive%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| eq_range_index_dive_limit | 200   |
+---------------------------+-------+
1 row in set (0.08 sec)
```

也就是说如果我们的 IN 语句中的参数个数小于200个的话，将使用 index dive 的方式计算各个单点区间对应的记录条数，如果大于或等于200个的话，可就不能使用 index dive 了，要使用所谓的索引统计数据来进行估算。怎么个估算法？继续往下看。

像会为每个表维护一份统计数据一样， MySQL 也会为表中的每一个索引维护一份统计数据，查看某个表中索引的统计数据可以使用 SHOW INDEX FROM 表名 的语法，比如我们查看一下 single_table 的各个索引的统计数据可以这么写：

```
mysql> SHOW INDEX FROM single_table;
+--------------+------------+--------------+--------------+-------------+-----------+-----
--------+----------+--------+------+------------+---------+---------------+
| Table        | Non_unique | Key_name     | Seq_in_index | Column_name | Collation | Card
inality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+--------------+------------+--------------+--------------+-------------+-----------+-----
--------+----------+--------+------+------------+---------+---------------+
| single_table |          0 | PRIMARY      |            1 | id          | A         |
9693 |     NULL | NULL   |      | BTREE      |         |               |
| single_table |          0 | idx_key2     |            1 | key2        | A         |
9693 |     NULL | NULL   | YES  | BTREE      |         |               |
| single_table |          1 | idx_key1     |            1 | key1        | A         |
968 |     NULL | NULL   | YES  | BTREE      |         |               |
| single_table |          1 | idx_key3     |            1 | key3        | A         |
799 |     NULL | NULL   | YES  | BTREE      |         |               |
| single_table |          1 | idx_key_part |            1 | key_part1   | A         |
9673 |     NULL | NULL   | YES  | BTREE      |         |               |
| single_table |          1 | idx_key_part |            2 | key_part2   | A         |
9999 |     NULL | NULL   | YES  | BTREE      |         |               |
| single_table |          1 | idx_key_part |            3 | key_part3   | A         |
10000 |     NULL | NULL   | YES  | BTREE      |         |               |
+--------------+------------+--------------+--------------+-------------+-----------+-----
--------+----------+--------+------+------------+---------+---------------+
7 rows in set (0.01 sec)
```

哇唔，竟然有这么多属性，不过好在这些属性都不难理解，我们就都介绍一遍吧：

属性名	描述
Table	索引所属表的名称。
Non_unique	索引列的值是否是唯一的，聚簇索引和唯一二级索引的该列值为 0 ，普通二级索引该列值为 1 。
Key_name	索引的名称。
Seq_in_index	索引列在索引中的位置，从1开始计数。比如对于联合索引 idx_key_part ，来说， key_part1 、 key_part2 和 key_part3 对应的位置分别是1、2、3。
Column_name	索引列的名称。
Collation	索引列中的值是按照何种排序方式存放的，值为 A 时代表升序存放，为 NULL 时代表降序存放。
Cardinality	索引列中不重复值的数量。后边我们会重点看这个属性的。
Sub_part	对于存储字符串或者字节串的列来说，有时候我们只想对这些串的前 n 个字符或字节建立索引，这个属性表示的就是那个 n 值。如果对完整的列建立索引的话，该属性的值就是 NULL 。
Packed	索引列如何被压缩， NULL 值表示未被压缩。这个属性我们暂时不了解，可以先忽略掉。
Null	该索引列是否允许存储 NULL 值。
Index_type	使用索引的类型，我们最常见的就是 BTREE ，其实也就是 B+ 树索引。
Comment	索引列注释信息。
Index_comment	索引注释信息。

上述属性除了 Packed 大家可能看不懂以外，应该没有啥看不懂的了，如果有的话肯定是大家看前边文章的时候跳过了啥东西。其实我们现在最在意的是 Cardinality 属性， Cardinality 直译过来就是 基数 的意思，表示索引列中不重复值的个数。比如对于一个一万行记录的表来说，某个索引列的 Cardinality 属性是 10000 ，那意味

着该列中没有重复的值，如果 Cardinality 属性是 1 的话，就意味着该列的值全部是重复的。不过需要注意的是，对于InnoDB存储引擎来说，使用SHOW INDEX语句展示出来的某个索引列的Cardinality属性是一个估计值，并不是精确的。关于这个 Cardinality 属性的值是如何被计算出来的我们后边再说，先看看它有什么用途。

前边说道，当 IN 语句中的参数个数大于或等于系统变量 eq_range_index_dive_limit 的值的话，就不会使用 index dive 的方式计算各个单点区间对应的索引记录条数，而是使用索引统计数据，这里所指的 索引统计数据 指的是这两个值：

- 使用 SHOW TABLE STATUS 展示出的 Rows 值，也就是一个表中有多少条记录。

  这个统计数据我们在前边唠叨全表扫描成本的时候说过很多遍了，就不赘述了。
- 使用 SHOW INDEX 语句展示出的 Cardinality 属性。

  结合上一个 Rows 统计数据，我们可以针对索引列，计算出平均一个值重复多少次。

  ```
  一个值的重复次数 ≈ Rows ÷ Cardinality
  ```

以 single_table 表的 idx_key1 索引为例，它的 Rows 值是 9693 ，它对应索引列 key1 的 Cardinality 值是 968 ，所以我们可以计算 key1 列平均单个值的重复次数就是：

```
9693 ÷ 968 ≈ 10（条）
```

此时再看上边那条查询语句：

```
SELECT * FROM single_table WHERE key1 IN ('aa1', 'aa2', 'aa3', ... , 'zzz');
```

假设 IN 语句中有20000个参数的话，就直接使用统计数据来估算这些参数需要单点区间对应的记录条数了，每个参数大约对应 10 条记录，所以总共需要回表的记录数就是：

```
20000 x 10 = 200000
```

使用统计数据来计算单点区间对应的索引记录条数可比 index dive 的方式简单多了，但是它的致命弱点就是：不精确！。使用统计数据算出来的查询成本与实际所需的成本可能相差非常大。

> 小贴士：
> 大家需要注意一下，在MySQL 5.7.3以及之前的版本中，eq_range_index_dive_limit的默认值为10，之后的版本默认值为200。所以如果大家采用的是5.7.3以及之前的版本的话，很容易采用索引统计数据而不是index dive的方式来计算查询成本。当你的查询中使用到了IN查询，但是却实际没有用到索引，就应该考虑一下是不是由于 eq_range_index_dive_limit 值太小导致的。

12.3 连接查询的成本

12.3.1 准备工作

连接查询至少是要有两个表的，只有一个 single_table 表是不够的，所以为了故事的顺利发展，我们直接构造一个和 single_table 表一模一样的 single_table2 表。为了简便起见，我们把 single_table 表称为 s1 表，把 single_table2 表称为 s2 表。

12.3.2 Condition filtering介绍

我们前边说过， MySQL 中连接查询采用的是嵌套循环连接算法，驱动表会被访问一次，被驱动表可能会被访问多次，所以对于两表连接查询来说，它的查询成本由下边两个部分构成：

- 单次查询驱动表的成本
- 多次查询被驱动表的成本（具体查询多少次取决于对驱动表查询的结果集中有多少条记录）

我们把对驱动表进行查询后得到的记录条数称之为驱动表的 扇出 （英文名： fanout ）。很显然驱动表的扇出值越小，对被驱动表的查询次数也就越少，连接查询的总成本也就越低。当查询优化器想计算整个连接查询所使用的成本时，就需要计算出驱动表的扇出值，有的时候扇出值的计算是很容易的，比如下边这两个查询：

- 查询一：

  ```
  SELECT * FROM single_table AS s1 INNER JOIN single_table2 AS s2;
  ```

  假设使用 s1 表作为驱动表，很显然对驱动表的单表查询只能使用全表扫描的方式执行，驱动表的扇出值也很明确，那就是驱动表中有多少记录，扇出值就是多少。我们前边说过，统计数据中 s1 表的记录行数是 9693 ，也就是说优化器就直接会把 9693 当作在 s1 表的扇出值。
- 查询二：

  ```
  SELECT * FROM single_table AS s1 INNER JOIN single_table2 AS s2
  WHERE s1.key2 >10 AND s1.key2 < 1000;
  ```

  仍然假设 s1 表是驱动表的话，很显然对驱动表的单表查询可以使用 idx_key2 索引执行查询。此时 idx_key2 的范围区间（10， 1000）中有多少条记录，那么扇出值就是多少。我们前边计算过，满足 idx_key2 的范围区间（10， 1000）的记录数是95条，也就是说本查询中优化器会把 95 当作驱动表 s1 的扇出值。

事情当然不会总是一帆风顺的，要不然剧情就太平淡了。有的时候扇出值的计算就变得很棘手，比方说下边几个查询：

- 查询三：

  ```
  SELECT * FROM single_table AS s1 INNER JOIN single_table2 AS s2
      WHERE s1.common_field > 'xyz';
  ```

  本查询和 查询一 类似，只不过对于驱动表 s1 多了一个 common_field > 'xyz' 的搜索条件。查询优化器又不会真正的去执行查询，所以它只能 猜 这 9693 记录里有多少条记录满足 common_field > 'xyz' 条件。
- 查询四：

  ```
  SELECT * FROM single_table AS s1 INNER JOIN single_table2 AS s2
      WHERE s1.key2 > 10 AND s1.key2 < 1000 AND
            s1.common_field > 'xyz';
  ```

  本查询和 查询二 类似，只不过对于驱动表 s1 也多了一个 common_field > 'xyz' 的搜索条件。不过因为本查询可以使用 idx_key2 索引，所以只需要从符合二级索引范围区间的记录中猜有多少条记录符合 common_field > 'xyz' 条件，也就是只需要猜在 95 条记录中有多少符合 common_field > 'xyz' 条件。
- 查询五：

  ```
  SELECT * FROM single_table AS s1 INNER JOIN single_table2 AS s2
      WHERE s1.key2 > 10 AND s1.key2 < 1000 AND
            s1.key1 IN ('a', 'b', 'c') AND
            s1.common_field > 'xyz';
  ```

  本查询和 查询二 类似，不过在驱动表 s1 选取 idx_key2 索引执行查询后，优化器需要从符合二级索引范围区间的记录中猜有多少条记录符合下边两个条件：
  - key1 IN ('a', 'b', 'c')
  - common_field > 'xyz'

  也就是优化器需要猜在 95 条记录中有多少符合上述两个条件的。

说了这么多，其实就是想表达在这两种情况下计算驱动表扇出值时需要靠 猜 ：

- 如果使用的是全表扫描的方式执行的单表查询，那么计算驱动表扇出时需要猜满足搜索条件的记录到底有多少条。
- 如果使用的是索引执行的单表扫描，那么计算驱动表扇出的时候需要猜满足除使用到对应索引的搜索条件外的其他搜索条件的记录有多少条。

设计 MySQL 的大叔把这个 猜 的过程称之为 condition filtering 。当然，这个过程可能会使用到索引，也可能使用到统计数据，也可能就是设计 MySQL 的大叔单纯的瞎猜，整个评估过程挺复杂的，再仔细的唠叨一遍可能引起大家的生理不适，所以我们就跳过了哈。

```
小贴士：
在MySQL 5.7之前的版本中，查询优化器在计算驱动表扇出时，如果是使用全表扫描的话，就直接使用表中记录的数量作为扇出值，如果使用索引的话，就直接使用满足范围条件的索引记录条数作为扇出值。在MySQL 5.7中，设计MySQL的大叔引入了这个condition filtering的功能，就是还要猜一猜剩余的那些搜索条件能把驱动表中的记录再过滤多少条，其实本质上就是为了让成本估算更精确。
我们所说的纯粹瞎猜其实是很不严谨的，设计MySQL的大叔们称之为启发式规则（heuristic），大家有兴趣的可以再深入了解一下哈。
```

12.3.3 两表连接的成本分析

连接查询的成本计算公式是这样的：

```
连接查询总成本 = 单次访问驱动表的成本 + 驱动表扇出数 x 单次访问被驱动表的成本
```

对于左（外）连接和右（外）连接查询来说，它们的驱动表是固定的，所以想要得到最优的查询方案只需要：

- 分别为驱动表和被驱动表选择成本最低的访问方法。

可是对于内连接来说，驱动表和被驱动表的位置是可以互换的，所以需要考虑两个方面的问题：

- 不同的表作为驱动表最终的查询成本可能是不同的，也就是需要考虑最优的表连接顺序。
- 然后分别为驱动表和被驱动表选择成本最低的访问方法。

很显然，计算内连接查询成本的方式更麻烦一些，下边我们就以内连接为例来看看如何计算出最优的连接查询方案。

```
小贴士：
左（外）连接和右（外）连接查询在某些特殊情况下可以被优化为内连接查询，我们在之后的章节中会仔细唠叨的，稍安勿躁。
```

比如对于下边这个查询来说：

```
SELECT * FROM single_table AS s1 INNER JOIN single_table2 AS s2
    ON s1.key1 = s2.common_field
    WHERE s1.key2 > 10 AND s1.key2 < 1000 AND
          s2.key2 > 1000 AND s2.key2 < 2000;
```

可以选择的连接顺序有两种：

- s1 连接 s2 ，也就是 s1 作为驱动表， s2 作为被驱动表。
- s2 连接 s1 ，也就是 s2 作为驱动表， s1 作为被驱动表。

查询优化器需要分别考虑这两种情况下的最优查询成本，然后选取那个成本更低的连接顺序以及该连接顺序下各个表的最优访问方法作为最终的查询计划。我们分别来看一下（定性的分析一下，不像分析单表查询那样定量的分析了）：

- 使用 s1 作为驱动表的情况

- 分析对于驱动表的成本最低的执行方案

  首先看一下涉及 s1 表单表的搜索条件有哪些：

  - s1.key2 > 10 AND s1.key2 < 1000

    所以这个查询可能使用到 idx_key2 索引，从全表扫描和使用 idx_key2 这两个方案中选出成本最低的那个，这个过程我们上边都唠叨过了，很显然使用 idx_key2 执行查询的成本更低些。

- 然后分析对于被驱动表的成本最低的执行方案

  此时涉及被驱动表 idx_key2 的搜索条件就是：

  - s2.common_field = 常数 （这是因为对驱动表 s1 结果集中的每一条记录，都需要进行一次被驱动表 s2 的访问，此时那些涉及两表的条件现在相当于只涉及被驱动表 s2 了。）
  - s2.key2 > 1000 AND s2.key2 < 2000

    很显然，第一个条件由于 common_field 没有用到索引，所以并没有什么卵用，此时访问 single_table2 表时可用的方案也是全表扫描和使用 idx_key2 两种，很显然使用 idx_key2 的成本更小。

  所以此时使用 single_table 作为驱动表时的总成本就是（暂时不考虑使用 join buffer 对成本的影响）：

  ```
  使用idx_key2访问s1的成本 + s1的扇出 × 使用idx_key2访问s2的成本
  ```

- 使用 s2 作为驱动表的情况
  - 分析对于驱动表的成本最低的执行方案

    首先看一下涉及 s2 表单表的搜索条件有哪些：

    - s2.key2 > 10 AND s2.key2 < 1000

      所以这个查询可能使用到 idx_key2 索引，从全表扫描和使用 idx_key2 这两个方案中选出成本最低的那个，这个过程我们上边都唠叨过了，很显然使用 idx_key2 执行查询的成本更低些。

  - 然后分析对于被驱动表的成本最低的执行方案

    此时涉及被驱动表 idx_key2 的搜索条件就是：

    - s1.key1 = 常数
    - s1.key2 > 1000 AND s1.key2 < 2000

      这时就很有趣了，使用 idx_key1 可以进行 ref 方式的访问，使用 idx_key2 可以使用 range 方式的访问。这是优化器需要从全表扫描、使用 idx_key1 、使用 idx_key2 这几个方案里选出一个成本最低的方案。这里有个问题啊，因为 idx_key2 的范围区间是确定的：(10, 1000) ，怎么计算使用 idx_key2 的成本我们上边已经说过了，可是在没有真正执行查询前， s1.key1 = 常数 中的 常数 值我们是不知道的，怎么衡量使用 idx_key1 执行查询的成本呢？其实很简单，直接使用索引统计数据就好了（就是索引列平均一个值重复多少次）。一般情况下， ref 的访问方式要比 range 成本最低，这里假设使用 idx_key1 进行对 s2 的访问。

    所以此时使用 single_table 作为驱动表时的总成本就是：

    ```
    使用idx_key2访问s2的成本 + s2的扇出 × 使用idx_key1访问s1的成本
    ```

最后优化器会比较这两种方式的最优访问成本，选取那个成本更低的连接顺序去真正的执行查询。从上边的计算过程也可以看出来，连接查询成本占大头的其实是 驱动表扇出数 x 单次访问被驱动表的成本 ，所以我们的优化重点其实是下边这两个部分：

- 尽量减少驱动表的扇出
- 对被驱动表的访问成本尽量低

这一点对于我们实际书写连接查询语句时十分有用，我们需要尽量在被驱动表的连接列上建立索引，这样就可以使用 ref 访问方法来降低访问被驱动表的成本了。如果可以，被驱动表的连接列最好是该表的主键或者唯一二级索引列，这样就可以把访问被驱动表的成本降到更低了。

12.3.4 多表连接的成本分析

首先要考虑一下多表连接时可能产生出多少种连接顺序：

- 对于两表连接，比如表A和表B连接

  只有 AB、BA这两种连接顺序。其实相当于 2 × 1 = 2 种连接顺序。
- 对于三表连接，比如表A、表B、表C进行连接

  有ABC、ACB、BAC、BCA、CAB、CBA这么6种连接顺序。其实相当于 3 × 2 × 1 = 6 种连接顺序。
- 对于四表连接的话，则会有 4 × 3 × 2 × 1 = 24 种连接顺序。
- 对于 n 表连接的话，则有 n × (n-1) × (n-2) × ··· × 1 种连接顺序，就是n的阶乘种连接顺序，也就是 n! 。

有 n 个表进行连接， MySQL 查询优化器要每一种连接顺序的成本都计算一遍么？那可是 n! 种连接顺序呀。其实真的是要都算一遍，不过设计 MySQL 的大叔们想了很多办法减少计算非常多种连接顺序的成本的方法：

- 提前结束某种顺序的成本评估

  MySQL 在计算各种链接顺序的成本之前，会维护一个全局的变量，这个变量表示当前最小的连接查询成本。如果在分析某个连接顺序的成本时，该成本已经超过当前最小的连接查询成本，那就压根儿不对该连接顺序继续往下分析了。比方说A、B、C三个表进行连接，已经得到连接顺序 ABC 是当前的最小连接成本，比方说 10.0 ，在计算连接顺序 BCA 时，发现 B 和 C 的连接成本就已经大于 10.0 时，就不再继续往后分析 BCA 这个连接顺序的成本了。
- 系统变量 optimizer_search_depth

  为了防止无穷无尽的分析各种连接顺序的成本，设计 MySQL 的大叔们提出了 optimizer_search_depth 系统变量，如果连接表的个数小于该值，那么就继续穷举分析每一种连接顺序的成本，否则只对与 optimizer_search_depth 值相同数量的表进行穷举分析。很显然，该值越大，成本分析的越精确，越容易得到好的执行计划，但是消耗的时间也就越长，否则得到不是很好的执行计划，但可以省掉很多分析连接成本的时间。
- 根据某些规则压根儿就不考虑某些连接顺序

  即使是有上边两条规则的限制，但是分析多个表不同连接顺序成本花费的时间还是会很长，所以设计 MySQL 的大叔干脆提出了一些所谓的 启发式规则 （就是根据以往经验指定的一些规则），凡是不满足这些规则的连接顺序压根儿就不分析，这样可以极大的减少需要分析的连接顺序的数量，但是也可能造成错失最优的执行计划。他们提供了一个系统变量 optimizer_prune_level 来控制到底是不是用这些启发式规则。

12.4 调节成本常数

我们前边之介绍了两个 成本常数 ：

- 读取一个页面花费的成本默认是 1.0
- 检测一条记录是否符合搜索条件的成本默认是 0.2

其实除了这两个成本常数， MySQL 还支持好多呢，它们被存储到了 mysql 数据库（这是一个系统数据库，我们之前介绍过）的两个表中：

```
mysql> SHOW TABLES FROM mysql LIKE '%cost%';
+--------------------------+
| Tables_in_mysql (%cost%) |
+--------------------------+
| engine_cost              |
| server_cost              |
+--------------------------+
2 rows in set (0.00 sec)
```

我们在第一章中就说过，一条语句的执行其实是分为两层的：

- server 层
- 存储引擎层

在 server 层进行连接管理、查询缓存、语法解析、查询优化等操作，在存储引擎层执行具体的数据存取操作。也就是说一条语句在 server 层中执行的成本是和它操作的表使用的存储引擎是没关系的，所以关于这些操作对应的 成本常数 就存储在了 server_cost 表中，而依赖于存储引擎的一些操作对应的 成本常数 就存储在了 engine_cost 表中。

12.4.1 mysql.server_cost表

server_cost 表中在 server 层进行的一些操作对应的 成本常数 ，具体内容如下：

```
mysql> SELECT * FROM mysql.server_cost;
+------------------------------+------------+---------------------+---------+
| cost_name                    | cost_value | last_update         | comment |
+------------------------------+------------+---------------------+---------+
| disk_temptable_create_cost   |       NULL | 2018-01-20 12:03:21 | NULL    |
| disk_temptable_row_cost      |       NULL | 2018-01-20 12:03:21 | NULL    |
| key_compare_cost             |       NULL | 2018-01-20 12:03:21 | NULL    |
| memory_temptable_create_cost |       NULL | 2018-01-20 12:03:21 | NULL    |
| memory_temptable_row_cost    |       NULL | 2018-01-20 12:03:21 | NULL    |
| row_evaluate_cost            |       NULL | 2018-01-20 12:03:21 | NULL    |
+------------------------------+------------+---------------------+---------+
6 rows in set (0.05 sec)
```

我们先看一下 server_cost 各个列都分别是什么意思：

- cost_name

  表示成本常数的名称。
- cost_value

  表示成本常数对应的值。如果该列的值为 NULL 的话，意味着对应的成本常数会采用默认值。
- last_update

  表示最后更新记录的时间。
- comment

  注释。

从 server_cost 中的内容可以看出来，目前在 server 层的一些操作对应的 成本常数 有以下几种：

成本常数名称	默认值	描述

成本常数名称	默认值	描述
disk_temptable_create_cost	40.0	创建基于磁盘的临时表的成本，如果增大这个值的话会让优化器尽量少的创建基于磁盘的临时表。
disk_temptable_row_cost	1.0	向基于磁盘的临时表写入或读取一条记录的成本，如果增大这个值的话会让优化器尽量少的创建基于磁盘的临时表。
key_compare_cost	0.1	两条记录做比较操作的成本，多用在排序操作上，如果增大这个值的话会提升 filesort 的成本，让优化器可能更倾向于使用索引完成排序而不是 filesort 。
memory_temptable_create_cost	2.0	创建基于内存的临时表的成本，如果增大这个值的话会让优化器尽量少的创建基于内存的临时表。
memory_temptable_row_cost	0.2	向基于内存的临时表写入或读取一条记录的成本，如果增大这个值的话会让优化器尽量少的创建基于内存的临时表。
row_evaluate_cost	0.2	这个就是我们之前一直使用的检测一条记录是否符合搜索条件的成本，增大这个值可能让优化器更倾向于使用索引而不是直接全表扫描。

> 小贴士：
> MySQL在执行诸如DISTINCT查询、分组查询、Union查询以及某些特殊条件下的排序查询都可能在内部先创建一个临时表，使用这个临时表来辅助完成查询（比如对于DISTINCT查询可以建一个带有UNIQUE索引的临时表，直接把需要去重的记录插入到这个临时表中，插入完成之后的记录就是结果集了）。在数据量大的情况下可能创建基于磁盘的临时表，也就是为该临时表使用MyISAM、InnoDB等存储引擎，在数据量不大时可能创建基于内存的临时表，也就是使用Memory存储引擎。关于更多临时表的细节我们并不打算展开唠叨，因为展开可能又需要好几万字了，大家知道创建临时表和对这个临时表进行写入和读取的操作代价还是很高的就行了。

这些成本常数在 server_cost 中的初始值都是 NULL ，意味着优化器会使用它们的默认值来计算某个操作的成本，如果我们想修改某个成本常数的值的话，需要做两个步骤：

- 对我们感兴趣的成本常数做更新操作

  比方说我们想把检测一条记录是否符合搜索条件的成本增大到 0.4 ，那么就可以这样写更新语句：

  ```
  UPDATE mysql.server_cost
      SET cost_value = 0.4
      WHERE cost_name = 'row_evaluate_cost';
  ```

- 让系统重新加载这个表的值。

  使用下边语句即可：

  ```
  FLUSH OPTIMIZER_COSTS;
  ```

当然，在你修改完某个成本常数后想把它们再改回默认值的话，可以直接把 cost_value 的值设置为 NULL ，再使用 FLUSH OPTIMIZER_COSTS 语句让系统重新加载它就好了。

12.4.2 mysql.engine_cost表

engine_cost表 表中在存储引擎层进行的一些操作对应的 成本常数 ，具体内容如下：

```
mysql> SELECT * FROM mysql.engine_cost;
+-------------+-------------+------------------------+------------+---------------------+-
--------+
| engine_name | device_type | cost_name              | cost_value | last_update         |
 comment |
+-------------+-------------+------------------------+------------+---------------------+-
--------+
| default     |           0 | io_block_read_cost     |       NULL | 2018-01-20 12:03:21 |
 NULL    |
| default     |           0 | memory_block_read_cost |       NULL | 2018-01-20 12:03:21 |
 NULL    |
+-------------+-------------+------------------------+------------+---------------------+-
--------+
2 rows in set (0.05 sec)
```

与 server_cost 相比， engine_cost 多了两个列：

- engine_name 列

  指成本常数适用的存储引擎名称。如果该值为 default ，意味着对应的成本常数适用于所有的存储引擎。
- device_type 列

  指存储引擎使用的设备类型，这主要是为了区分常规的机械硬盘和固态硬盘，不过在 MySQL 5.7.21 这个版本中并没有对机械硬盘的成本和固态硬盘的成本作区分，所以该值默认是 0 。

我们从 engine_cost 表中的内容可以看出来，目前支持的存储引擎成本常数只有两个：

成本常数名称	默认值	描述
io_block_read_cost	1.0	从磁盘上读取一个块对应的成本。请注意我使用的是 块 ，而不是 页 这个词儿。对于 InnoDB 存储引擎来说，一个 页 就是一个块，不过对于 MyISAM 存储引擎来说，默认是以 4096 字节作为一个块的。增大这个值会加重 I/O 成本，可能让优化器更倾向于选择使用索引执行查询而不是执行全表扫描。
memory_block_read_cost	1.0	与上一个参数类似，只不过衡量的是从内存中读取一个块对应的成本。

大家看完这两个成本常数的默认值是不是有些疑惑，怎么从内存中和从磁盘上读取一个块的默认成本是一样的，脑子瓦特了？这主要是因为在 MySQL 目前的实现中，并不能准确预测某个查询需要访问的块中有哪些块已经加载到内存中，有哪些块还停留在磁盘上，所以设计 MySQL 的大叔们很粗暴的认为不管这个块有没有加载到内存中，使用的成本都是 1.0 ，不过随着 MySQL 的发展，等到可以准确预测哪些块在磁盘上，那些块在内存中的那一天，这两个成本常数的默认值可能会改一改吧。

与更新 server_cost 表中的记录一样，我们也可以通过更新 engine_cost 表中的记录来更改关于存储引擎的成本常数，我们也可以通过为 engine_cost 表插入新记录的方式来添加只针对某种存储引擎的成本常数：

- 插入针对某个存储引擎的成本常数

  比如我们想增大 InnoDB 存储引擎页面 I/O 的成本，书写正常的插入语句即可：

  ```
  INSERT INTO mysql.engine_cost
      VALUES ('InnoDB', 0, 'io_block_read_cost', 2.0,
      CURRENT_TIMESTAMP, 'increase Innodb I/O cost');
  ```

- 让系统重新加载这个表的值。

  使用下边语句即可：

```
FLUSH OPTIMIZER_COSTS;
```

13 第13章 兵马未动，粮草先行-InnoDB统计数据是如何收集的

标签： MySQL 是怎样运行的

我们前边唠叨查询成本的时候经常用到一些统计数据，比如通过 SHOW TABLE STATUS 可以看到关于表的统计数据，通过 SHOW INDEX 可以看到关于索引的统计数据，那么这些统计数据是怎么来的呢？它们是以什么方式收集的呢？本章将聚焦于 InnoDB 存储引擎的统计数据收集策略，看完本章大家就会明白为啥前边老说 InnoDB 的统计信息是不精确的估计值了（言下之意就是我们不打算介绍 MyISAM 存储引擎统计数据的收集和存储方式，有想了解的同学自己个儿看看文档哈）。

13.1 两种不同的统计数据存储方式

InnoDB 提供了两种存储统计数据的方式：

- 永久性的统计数据

  这种统计数据存储在磁盘上，也就是服务器重启之后这些统计数据还在。
- 非永久性的统计数据

  这种统计数据存储在内存中，当服务器关闭时这些这些统计数据就都被清除掉了，等到服务器重启之后，在某些适当的场景下才会重新收集这些统计数据。

设计 MySQL 的大叔们给我们提供了系统变量 innodb_stats_persistent 来控制到底采用哪种方式去存储统计数据。在 MySQL 5.6.6 之前， innodb_stats_persistent 的值默认是 OFF ，也就是说 InnoDB 的统计数据默认是存储到内存的，之后的版本中 innodb_stats_persistent 的值默认是 ON ，也就是统计数据默认被存储到磁盘中。

不过 InnoDB 默认是以表为单位来收集和存储统计数据的，也就是说我们可以把某些表的统计数据（以及该表的索引统计数据）存储在磁盘上，把另一些表的统计数据存储在内存中。怎么做到的呢？我们可以在创建和修改表的时候通过指定 STATS_PERSISTENT 属性来指明该表的统计数据存储方式：

```
CREATE TABLE 表名 (...) Engine=InnoDB, STATS_PERSISTENT = (1|0);

ALTER TABLE 表名 Engine=InnoDB, STATS_PERSISTENT = (1|0);
```

当 STATS_PERSISTENT=1 时，表明我们想把该表的统计数据永久的存储到磁盘上，当 STATS_PERSISTENT=0 时，表明我们想把该表的统计数据临时的存储到内存中。如果我们在创建表时未指定 STATS_PERSISTENT 属性，那默认采用系统变量 innodb_stats_persistent 的值作为该属性的值。

13.2 基于磁盘的永久性统计数据

当我们选择把某个表以及该表索引的统计数据存放到磁盘上时，实际上是把这些统计数据存储到了两个表里：

```
mysql> SHOW TABLES FROM mysql LIKE 'innodb%';
+---------------------------+
| Tables_in_mysql (innodb%) |
+---------------------------+
| innodb_index_stats        |
| innodb_table_stats        |
+---------------------------+
2 rows in set (0.01 sec)
```

可以看到，这两个表都位于 mysql 系统数据库下边，其中：

- innodb_table_stats 存储了关于表的统计数据，每一条记录对应着一个表的统计数据。
- innodb_index_stats 存储了关于索引的统计数据，每一条记录对应着一个索引的一个统计项的统计数据。

我们下边的任务就是看一下这两个表里边都有什么以及表里的数据是如何生成的。

13.2.1 innodb_table_stats

直接看一下这个 innodb_table_stats 表中的各个列都是干嘛的：

字段名	描述
database_name	数据库名
table_name	表名
last_update	本条记录最后更新时间
n_rows	表中记录的条数
clustered_index_size	表的聚簇索引占用的页面数量
sum_of_other_index_sizes	表的其他索引占用的页面数量

注意这个表的主键是 (database_name,table_name) ，也就是innodb_table_stats表的每条记录代表着一个表的统计信息。我们直接看一下这个表里的内容：

```
mysql> SELECT * FROM mysql.innodb_table_stats;
+---------------+---------------+---------------------+--------+----------------------+---
-----------------------+
| database_name | table_name    | last_update         | n_rows | clustered_index_size | su
m_of_other_index_sizes |
+---------------+---------------+---------------------+--------+----------------------+---
-----------------------+
| mysql         | gtid_executed | 2018-07-10 23:51:36 |      0 |                    1 |
0 |
| sys           | sys_config    | 2018-07-10 23:51:38 |      5 |                    1 |
0 |
| xiaohaizi     | single_table  | 2018-12-10 17:03:13 |   9693 |                   97 |
175 |
+---------------+---------------+---------------------+--------+----------------------+---
-----------------------+
3 rows in set (0.01 sec)
```

可以看到我们熟悉的 single_table 表的统计信息就对应着 mysql.innodb_table_stats 的第三条记录。几个重要统计信息项的值如下：

- `n_rows` 的值是 `9693` ，表明 `single_table` 表中大约有 `9693` 条记录，注意这个数据是估计值。
- `clustered_index_size` 的值是 `97` ，表明 `single_table` 表的聚簇索引占用97个页面，这个值是也是一个估计值。
- `sum_of_other_index_sizes` 的值是 `175` ，表明 `single_table` 表的其他索引一共占用175个页面，这个值是也是一个估计值。

13.2.1.1 n_rows统计项的收集

为啥老强调 `n_rows` 这个统计项的值是估计值呢？现在就来揭晓答案。 `InnoDB` 统计一个表中有多少行记录的套路是这样的：

- 按照一定算法（并不是纯粹随机的）选取几个叶子节点页面，计算每个页面中主键值记录数量，然后计算平均一个页面中主键值的记录数量乘以全部叶子节点的数量就算是该表的 `n_rows` 值。

  > 小贴士：
  > 真实的计算过程比这个稍微复杂一些，不过大致上就是这样的啦～

  可以看出来这个 `n_rows` 值精确与否取决于统计时采样的页面数量，设计 `MySQL` 的大叔很贴心的为我们准备了一个名为 `innodb_stats_persistent_sample_pages` 的系统变量来控制使用永久性的统计数据时，计算统计数据时采样的页面数量。该值设置的越大，统计出的 `n_rows` 值越精确，但是统计耗时也就最久；该值设置的越小，统计出的 `n_rows` 值越不精确，但是统计耗时特别少。所以在实际使用是需要我们去权衡利弊，该系统变量的默认值是 `20` 。

  我们前边说过，不过 `InnoDB` 默认是以表为单位来收集和存储统计数据的，我们也可以单独设置某个表的采样页面的数量，设置方式就是在创建或修改表的时候通过指定 `STATS_SAMPLE_PAGES` 属性来指明该表的统计数据存储方式：

  ```
  CREATE TABLE 表名 (...) Engine=InnoDB, STATS_SAMPLE_PAGES = 具体的采样页面数量;

  ALTER TABLE 表名 Engine=InnoDB, STATS_SAMPLE_PAGES = 具体的采样页面数量;
  ```

  如果我们在创建表的语句中并没有指定 `STATS_SAMPLE_PAGES` 属性的话，将默认使用系统变量 `innodb_stats_persistent_sample_pages` 的值作为该属性的值。

13.2.1.2 clustered_index_size和sum_of_other_index_sizes统计项的收集

统计这两个数据需要大量用到我们之前唠叨的 `InnoDB` 表空间的知识，如果大家压根儿没有看那一章，那下边的计算过程大家还是不要看了（看也看不懂）；如果看过了，那大家就会发现 `InnoDB` 表空间的知识真是有用啊啊啊！！！

这两个统计项的收集过程如下：

- 从数据字典里找到表的各个索引对应的根页面位置。

  系统表 `SYS_INDEXES` 里存储了各个索引对应的根页面信息。
- 从根页面的 `Page Header` 里找到叶子节点段和非叶子节点段对应的 `Segment Header` 。

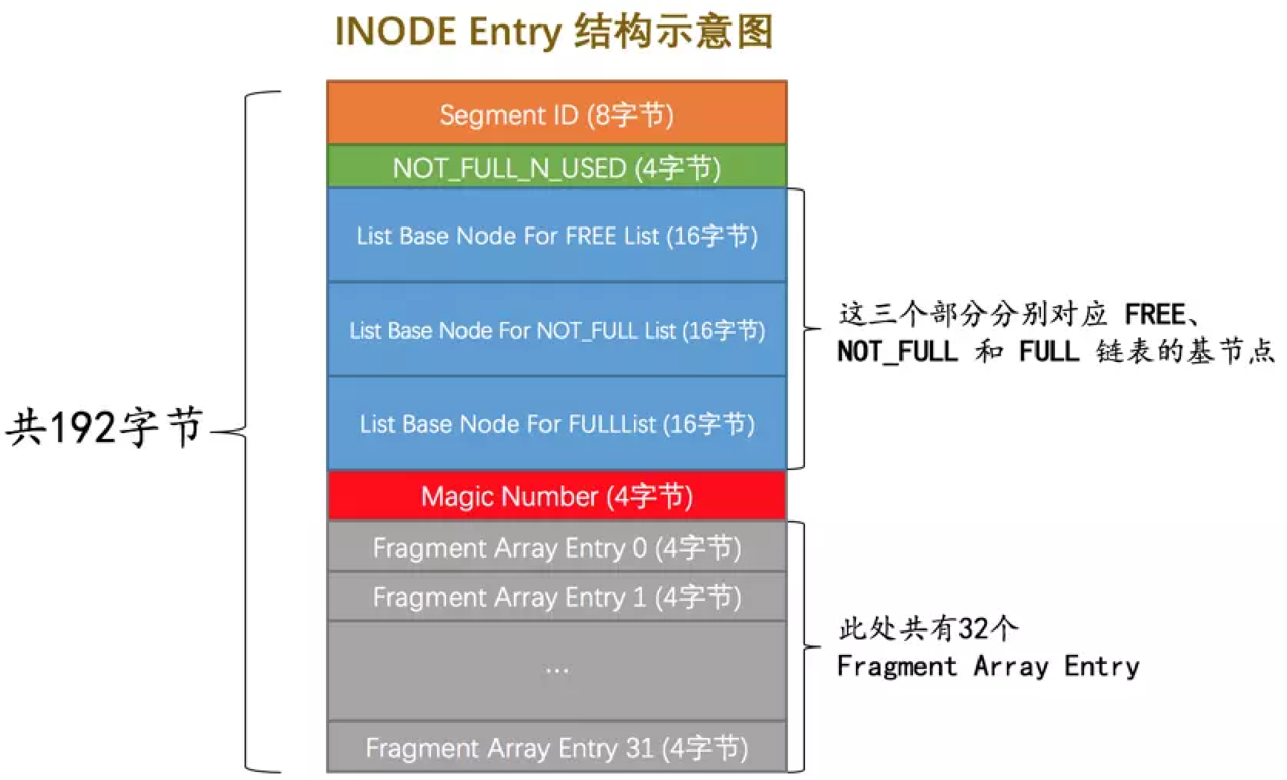
  在每个索引的根页面的 `Page Header` 部分都有两个字段：
  - `PAGE_BTR_SEG_LEAF` ：表示B+树叶子段的 `Segment Header` 信息。
  - `PAGE_BTR_SEG_TOP` ：表示B+树非叶子段的 `Segment Header` 信息。
- 从叶子节点段和非叶子节点段的 `Segment Header` 中找到这两个段对应的 `INODE Entry` 结构。

  这个是 `Segment Header` 结构：



- 从对应的 INODE Entry 结构中可以找到该段对应所有零散的页面地址以及 FREE 、 NOT_FULL 、 FULL 链表的基节点。

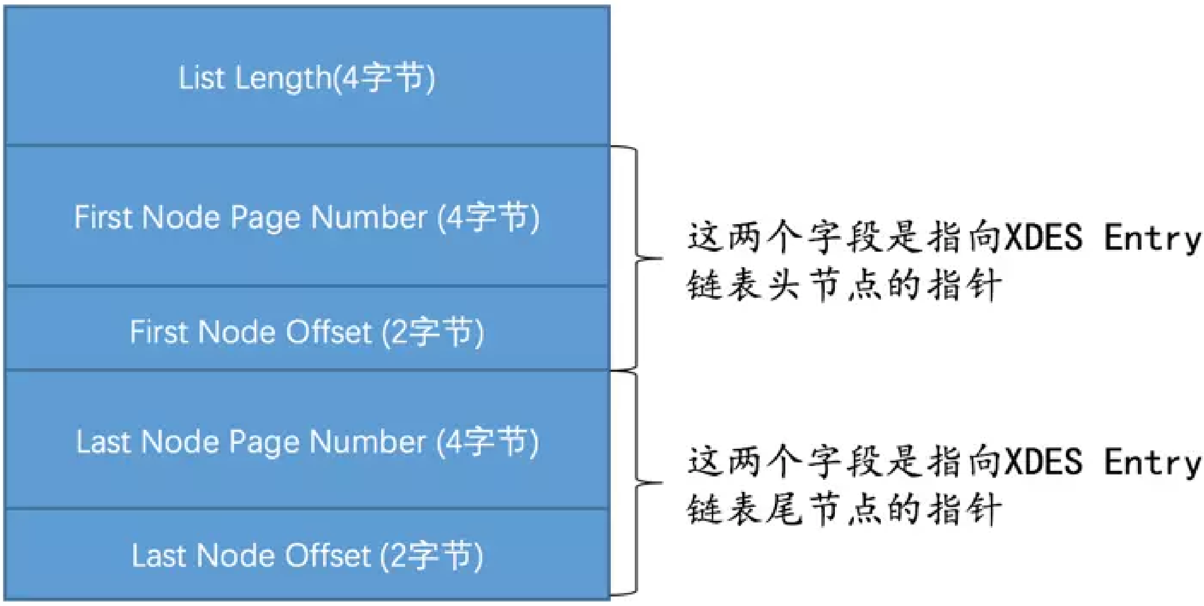
  这个是 INODE Entry 结构：



- 直接统计零散的页面有多少个，然后从那三个链表的 List Length 字段中读出该段占用的区的大小，每个区占用 64 个页，所以就可以统计出整个段占用的页面。

  这个是链表基节点的示意图：

List Base Node 结构示意图



- 分别计算聚簇索引的叶子结点段和非叶子节点段占用的页面数，它们的和就是 clustered_index_size 的值，按照同样的套路把其余索引占用的页面数都算出来，加起来之后就是 sum_of_other_index_sizes 的值。

这里需要大家注意一个问题，我们说一个段的数据在非常多时（超过32个页面），会以 区 为单位来申请空间，这里头的问题是以区为单位申请空间中有一些页可能并没有使用，但是在统计 clustered_index_size 和 sum_of_other_index_sizes 时都把它们算进去了，所以说聚簇索引和其他的索引占用的页面数可能比这两个值要小一些。

13.2.2 innodb_index_stats

直接看一下这个 innodb_index_stats 表中的各个列都是干嘛的：

字段名	描述
database_name	数据库名
table_name	表名
index_name	索引名
last_update	本条记录最后更新时间
stat_name	统计项的名称
stat_value	对应的统计项的值
sample_size	为生成统计数据而采样的页面数量
stat_description	对应的统计项的描述

注意这个表的主键是（database_name, table_name, index_name, stat_name），其中的 stat_name 是指统计项的名称，也就是说innodb_index_stats表的每条记录代表着一个索引的一个统计项。可能这会大家有些懵逼这个统计项到底指什么，别着急，我们直接看一下关于 single_table 表的索引统计数据都有些什么：

```
mysql> SELECT * FROM mysql.innodb_index_stats WHERE table_name = 'single_table';
+---------------+--------------+--------------+---------------------+--------------+------
------+-------------+-----------------------------------+
| database_name | table_name   | index_name   | last_update         | stat_name    | stat_
value | sample_size | stat_description                  |
+---------------+--------------+--------------+---------------------+--------------+------
------+-------------+-----------------------------------+
| xiaohaizi     | single_table | PRIMARY      | 2018-12-14 14:24:46 | n_diff_pfx01 |
9693 |          20 | id                                |
| xiaohaizi     | single_table | PRIMARY      | 2018-12-14 14:24:46 | n_leaf_pages |
91 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | PRIMARY      | 2018-12-14 14:24:46 | size         |
97 |        NULL | Number of pages in the index      |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | n_diff_pfx01 |
968 |          28 | key1                              |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | n_diff_pfx02 |
 10000 |          28 | key1,id                           |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | n_leaf_pages |
28 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | size         |
29 |        NULL | Number of pages in the index      |
| xiaohaizi     | single_table | idx_key2     | 2018-12-14 14:24:46 | n_diff_pfx01 |
 10000 |          16 | key2                              |
| xiaohaizi     | single_table | idx_key2     | 2018-12-14 14:24:46 | n_leaf_pages |
16 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | idx_key2     | 2018-12-14 14:24:46 | size         |
17 |        NULL | Number of pages in the index      |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | n_diff_pfx01 |
799 |          31 | key3                              |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | n_diff_pfx02 |
 10000 |          31 | key3,id                           |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | n_leaf_pages |
31 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | size         |
32 |        NULL | Number of pages in the index      |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx01 |
9673 |          64 | key_part1                         |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx02 |
9999 |          64 | key_part1,key_part2               |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx03 |
 10000 |          64 | key_part1,key_part2,key_part3     |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx04 |
 10000 |          64 | key_part1,key_part2,key_part3,id  |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_leaf_pages |
64 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | size         |
97 |        NULL | Number of pages in the index      |
+---------------+--------------+--------------+---------------------+--------------+------
------+-------------+-----------------------------------+
20 rows in set (0.03 sec)
```

这个结果有点儿多，正确查看这个结果的方式是这样的：

- 先查看 index_name 列，这个列说明该记录是哪个索引的统计信息，从结果中我们可以看出来， PRIMARY 索引（也就是主键）占了3条记录， idx_key_part 索引占了6条记录。
- 针对 index_name 列相同的记录， stat_name 表示针对该索引的统计项名称， stat_value 展示的是该索引在该统计项上的值， stat_description 指的是来描述该统计项的含义的。我们来具体看一下一个索引都有哪些统计项：
  - n_leaf_pages ： 表示该索引的叶子节点占用多少页面。
  - size ： 表示该索引共占用多少页面。
  - n_diff_pfxNN ： 表示对应的索引列不重复的值有多少。其中的 NN 长得有点儿怪呀，啥意思呢？

    其实 NN 可以被替换为 01 、 02 、 03 ... 这样的数字。比如对于 idx_key_part 来说：
    - n_diff_pfx01 表示的是统计 key_part1 这单单一个列不重复的值有多少。
    - n_diff_pfx02 表示的是统计 key_part1、key_part2 这两个列组合起来不重复的值有多少。
    - n_diff_pfx03 表示的是统计 key_part1、key_part2、key_part3 这三个列组合起来不重复的值有多少。
    - n_diff_pfx04 表示的是统计 key_part1、key_part2、key_part3、id 这四个列组合起来不重复的值有多少。

      小贴士：
      这里需要注意的是，对于普通的二级索引，并不能保证它的索引列值是唯一的，比如对于idx_key1来说，key1列就可能有很多值重复的记录。此时只有在索引列上加上主键值才可以区分两条索引列值都一样的二级索引记录。对于主键和唯一二级索引则没有这个问题，它们本身就可以保证索引列值的不重复，所以也不需要再统计一遍在索引列后加上主键值的不重复值有多少。比如上边的idx_key1有n_diff_pfx01、n_diff_pfx02两个统计项，而idx_key2却只有n_diff_pfx01一个统计项。

- 在计算某些索引列中包含多少不重复值时，需要对一些叶子节点页面进行采样， size 列就表明了采样的页面数量是多少。

  小贴士：
  对于有多个列的联合索引来说，采样的页面数量是：innodb_stats_persistent_sample_pages × 索引列的个数。当需要采样的页面数量大于该索引的叶子节点数量的话，就直接采用全表扫描来统计索引列的不重复值数量了。所以大家可以在查询结果中看到不同索引对应的size列的值可能是不同的。

13.2.3 定期更新统计数据

随着我们不断的对表进行增删改操作，表中的数据也一直在变化， innodb_table_stats 和 innodb_index_stats 表里的统计数据是不是也应该跟着变一变了？当然要变了，不变的话 MySQL 查询优化器计算的成本可就差老鼻子远了。设计 MySQL 的大叔提供了如下两种更新统计数据的方式：

- 开启 innodb_stats_auto_recalc 。

  系统变量 innodb_stats_auto_recalc 决定着服务器是否自动重新计算统计数据，它的默认值是 ON ，也就是该功能默认是开启的。每个表都维护了一个变量，该变量记录着对该表进行增删改的记录条数，如果发生变动的记录数量超过了表大小的 10% ，并且自动重新计算统计数据的功能是打开的，那么服务器会重新进行一次统计数据的计算，并且更新 innodb_table_stats 和 innodb_index_stats 表。不过自动重新计算统计数据的过程是异步发生的，也就是即使表中变动的记录数超过了 10% ，自动重新计算统计数据也不会立即发生，可能会延迟几秒才会进行计算。

  再一次强调， InnoDB 默认是以表为单位来收集和存储统计数据的，我们也可以单独为某个表设置是否自动重新计算统计数的属性，设置方式就是在创建或修改表的时候通过指定 STATS_AUTO_RECALC 属性来指明该表的统计数据存储方式：

```
CREATE TABLE 表名 (...) Engine=InnoDB, STATS_AUTO_RECALC = (1|0);

ALTER TABLE 表名 Engine=InnoDB, STATS_AUTO_RECALC = (1|0);
```

当 STATS_AUTO_RECALC=1 时，表明我们想让该表自动重新计算统计数据，当 STATS_PERSISTENT=0 时，表明不想让该表自动重新计算统计数据。如果我们在创建表时未指定 STATS_AUTO_RECALC 属性，那默认采用系统变量 innodb_stats_auto_recalc 的值作为该属性的值。

- 手动调用 ANALYZE TABLE 语句来更新统计信息

  如果 innodb_stats_auto_recalc 系统变量的值为 OFF 的话，我们也可以手动调用 ANALYZE TABLE 语句来重新计算统计数据，比如我们可以这样更新关于 single_table 表的统计数据：

```
mysql> ANALYZE TABLE single_table;
+------------------------+---------+----------+----------+
| Table                  | Op      | Msg_type | Msg_text |
+------------------------+---------+----------+----------+
| xiaohaizi.single_table | analyze | status   | OK       |
+------------------------+---------+----------+----------+
1 row in set (0.08 sec)
```

  需要注意的是，ANALYZE TABLE语句会立即重新计算统计数据，也就是这个过程是同步的，在表中索引多或者采样页面特别多时这个过程可能会特别慢，请不要没事儿就运行一下 ANALYZE TABLE 语句，最好在业务不是很繁忙的时候再运行。

13.2.4 手动更新 innodb_table_stats 和 innodb_index_stats 表

其实 innodb_table_stats 和 innodb_index_stats 表就相当于一个普通的表一样，我们能对它们做增删改查操作。这也就意味着我们可以手动更新某个表或者索引的统计数据。比如说我们想把 single_table 表关于行数的统计数据更改一下可以这么做：

- 步骤一：更新 innodb_table_stats 表。

```
UPDATE innodb_table_stats
    SET n_rows = 1
    WHERE table_name = 'single_table';
```

- 步骤二：让 MySQL 查询优化器重新加载我们更改过的数据。

  更新完 innodb_table_stats 只是单纯的修改了一个表的数据，需要让 MySQL 查询优化器重新加载我们更改过的数据，运行下边的命令就可以了：

```
FLUSH TABLE single_table;
```

之后我们使用 SHOW TABLE STATUS 语句查看表的统计数据时就看到 Rows 行变为了 1 。

13.3 基于内存的非永久性统计数据

当我们把系统变量 innodb_stats_persistent 的值设置为 OFF 时，之后创建的表的统计数据默认就都是非永久性的了，或者我们直接在创建表或修改表时设置 STATS_PERSISTENT 属性的值为 0 ，那么该表的统计数据就是非永久性的了。

与永久性的统计数据不同，非永久性的统计数据采样的页面数量是由 innodb_stats_transient_sample_pages 控制的，这个系统变量的默认值是 8 。

另外，由于非永久性的统计数据经常更新，所以导致 MySQL 查询优化器计算查询成本的时候依赖的是经常变化的统计数据，也就会生成经常变化的执行计划，这个可能让大家有些懵逼。不过最近的 MySQL 版本都不咋用这种基于内存的非永久性统计数据了，所以我们也就不深入唠叨它了。

13.4 innodb_stats_method的使用

我们知道 索引列不重复的值的数量 这个统计数据对于 MySQL 查询优化器十分重要，因为通过它可以计算出在索引列中平均一个值重复多少行，它的应用场景主要有两个：

- 单表查询中单点区间太多，比方说这样：

  ```
  SELECT * FROM tbl_name WHERE key IN ('xx1', 'xx2', ..., 'xxn');
  ```

  当 IN 里的参数数量过多时，采用 index dive 的方式直接访问 B+ 树索引去统计每个单点区间对应的记录的数量就太耗费性能了，所以直接依赖统计数据中的平均一个值重复多少行来计算单点区间对应的记录数量。
- 连接查询时，如果有涉及两个表的等值匹配连接条件，该连接条件对应的被驱动表中的列又拥有索引时，则可以使用 ref 访问方法来对被驱动表进行查询，比方说这样：

  ```
  SELECT * FROM t1 JOIN t2 ON t1.column = t2.key WHERE ...;
  ```

  在真正执行对 t2 表的查询前， t1.comumn 的值是不确定的，所以我们也不能通过 index dive 的方式直接访问 B+ 树索引去统计每个单点区间对应的记录的数量，所以也只能依赖统计数据中的平均一个值重复多少行来计算单点区间对应的记录数量。

在统计索引列不重复的值的数量时，有一个比较烦的问题就是索引列中出现 NULL 值怎么办，比方说某个索引列的内容是这样：

```
+------+
| col  |
+------+
|    1 |
|    2 |
| NULL |
| NULL |
+------+
```

此时计算这个 col 列中不重复的值的数量就有下边的分歧：

- 有的人认为 NULL 值代表一个未确定的值，所以设计 MySQL 的大叔才认为任何和 NULL 值做比较的表达式的值都为 NULL ，就是这样：

```
mysql> SELECT 1 = NULL;
+----------+
| 1 = NULL |
+----------+
|     NULL |
+----------+
1 row in set (0.00 sec)

mysql> SELECT 1 != NULL;
+-----------+
| 1 != NULL |
+-----------+
|      NULL |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT NULL = NULL;
+-------------+
| NULL = NULL |
+-------------+
|        NULL |
+-------------+
1 row in set (0.00 sec)

mysql> SELECT NULL != NULL;
+--------------+
| NULL != NULL |
+--------------+
|         NULL |
+--------------+
1 row in set (0.00 sec)
```

所以每一个`NULL`值都是独一无二的，也就是说统计索引列不重复的值的数量时，应该把`NULL`值当作一个独立的值，所以`col`列的不重复的值的数量就是：`4`（分别是1、2、NULL、NULL这四个值）。

- 有的人认为其实 `NULL` 值在业务上就是代表没有，所有的 `NULL` 值代表的意义是一样的，所以 `col` 列不重复的值的数量就是： `3` （分别是1、2、NULL这三个值）。
- 有的人认为这 `NULL` 完全没有意义嘛，所以在统计索引列不重复的值的数量时压根儿不能把它们算进来，所以 `col` 列不重复的值的数量就是： `2` （分别是1、2这两个值）。

设计 `MySQL` 的大叔蛮贴心的，他们提供了一个名为 `innodb_stats_method` 的系统变量，相当于在计算某个索引列不重复值的数量时如何对待 `NULL` 值这个锅甩给了用户，这个系统变量有三个候选值：

- `nulls_equal` ：认为所有 `NULL` 值都是相等的。这个值也是 `innodb_stats_method` 的默认值。

  如果某个索引列中 `NULL` 值特别多的话，这种统计方式会让优化器认为某个列中平均一个值重复次数特别多，所以倾向于不使用索引进行访问。
- `nulls_unequal` ：认为所有 `NULL` 值都是不相等的。

  如果某个索引列中 `NULL` 值特别多的话，这种统计方式会让优化器认为某个列中平均一个值重复次数特别少，所以倾向于使用索引进行访问。

- nulls_ignored ：直接把 NULL 值忽略掉。

反正这个锅是甩给用户了，当你选定了 innodb_stats_method 值之后，优化器即使选择了不是最优的执行计划，那也跟设计 MySQL 的大叔们没关系了哈～ 当然对于用户的我们来说，最好不在索引列中存放NULL值才是正解。

13.5 总结

- InnoDB 以表为单位来收集统计数据，这些统计数据可以是基于磁盘的永久性统计数据，也可以是基于内存的非永久性统计数据。
- innodb_stats_persistent 控制着使用永久性统计数据还是非永久性统计数据；innodb_stats_persistent_sample_pages 控制着永久性统计数据的采样页面数量；innodb_stats_transient_sample_pages 控制着非永久性统计数据的采样页面数量；innodb_stats_auto_recalc 控制着是否自动重新计算统计数据。
- 我们可以针对某个具体的表，在创建和修改表时通过指定 STATS_PERSISTENT 、 STATS_AUTO_RECALC 、STATS_SAMPLE_PAGES 的值来控制相关统计数据属性。
- innodb_stats_method 决定着在统计某个索引列不重复值的数量时如何对待 NULL 值。

14 第14章 不好看就要多整容-MySQL基于规则的优化（内含关于子查询优化二三事儿）

标签： MySQL 是怎样运行的

大家别忘了 MySQL 本质上是一个软件，设计 MySQL 的大叔并不能要求使用这个软件的人个个都是数据库高高手，就像我写这本书的时候并不能要求各位在学之前就会了里边儿的知识。

> 吐槽一下：都会了的人谁还看呢，难道是为了精神上受感化？

也就是说我们无法避免某些同学写一些执行起来十分耗费性能的语句。即使是这样，设计 MySQL 的大叔还是依据一些规则，竭尽全力的把这个很糟糕的语句转换成某种可以比较高效执行的形式，这个过程也可以被称作 查询重写 （就是人家觉得你写的语句不好，自己再重写一遍）。本章详细唠叨一下一些比较重要的重写规则。

14.1 条件化简

我们编写的查询语句的搜索条件本质上是一个表达式，这些表达式可能比较繁杂，或者不能高效的执行， MySQL 的查询优化器会为我们简化这些表达式。为了方便大家理解，我们后边举例子的时候都使用诸如 a 、 b 、 c 之类的简单字母代表某个表的列名。

14.1.1 移除不必要的括号

有时候表达式里有许多无用的括号，比如这样：

```
((a = 5 AND b = c) OR ((a > c) AND (c < 5)))
```

看着就很烦，优化器会把那些用不到的括号给干掉，就是这样：

```
(a = 5 and b = c) OR (a > c AND c < 5)
```

14.1.2 常量传递（constant_propagation）

有时候某个表达式是某个列和某个常量做等值匹配，比如这样：

```
a = 5
```

当这个表达式和其他涉及列 a 的表达式使用 AND 连接起来时，可以将其他表达式中的 a 的值替换为 5 ，比如这样：

```
a = 5 AND b > a
```

就可以被转换为：

```
a = 5 AND b > 5
```

> 小贴士：
> 为啥用OR连接起来的表达式就不能进行常量传递呢？自己想想哈～

14.1.3 等值传递（equality_propagation）

有时候多个列之间存在等值匹配的关系，比如这样：

```
a = b and b = c and c = 5
```

这个表达式可以被简化为：

```
a = 5 and b = 5 and c = 5
```

14.1.4 移除没用的条件（trivial_condition_removal）

对于一些明显永远为 TRUE 或者 FALSE 的表达式，优化器会移除掉它们，比如这个表达式：

```
(a < 1 and b = b) OR (a = 6 OR 5 != 5)
```

很明显， b = b 这个表达式永远为 TRUE ， 5 != 5 这个表达式永远为 FALSE ，所以简化后的表达式就是这样的：

```
(a < 1 and TRUE) OR (a = 6 OR FALSE)
```

可以继续被简化为

```
a < 1 OR a = 6
```

14.1.5 表达式计算

在查询开始执行之前，如果表达式中只包含常量的话，它的值会被先计算出来，比如这个：

```
a = 5 + 1
```

因为 5 + 1 这个表达式只包含常量，所以就会被化简成：

```
a = 6
```

但是这里需要注意的是，如果某个列并不是以单独的形式作为表达式的操作数时，比如出现在函数中，出现在某个更复杂表达式中，就像这样：

```
ABS(a) > 5
```

或者：

```
-a < -8
```

优化器是不会尝试对这些表达式进行化简的。我们前边说过只有搜索条件中索引列和常数使用某些运算符连接起来才可能使用到索引，所以如果可以的话，最好让索引列以单独的形式出现在表达式中。

14.1.6 HAVING子句和WHERE子句的合并

如果查询语句中没有出现诸如 SUM 、 MAX 等等的聚集函数以及 GROUP BY 子句，优化器就把 HAVING 子句和 WHERE 子句合并起来。

14.1.7 常量表检测

设计 MySQL 的大叔觉得下边这两种查询运行的特别快：

- 查询的表中一条记录没有，或者只有一条记录。

> 小贴士：
> 大家有没有觉得这一条有点儿不对劲，我还没开始查表呢咋就知道这表里边有几条记录呢？哈哈，这个其实依靠的是统计数据。不过我们说过InnoDB的统计数据数据不准确，所以这一条不能用于使用InnoDB作为存储引擎的表，只能适用于使用Memory或者MyISAM存储引擎的表。

- 使用主键等值匹配或者唯一二级索引列等值匹配作为搜索条件来查询某个表。

设计 MySQL 的大叔觉得这两种查询花费的时间特别少，少到可以忽略，所以也把通过这两种方式查询的表称之为 常量表 （英文名： constant tables ）。优化器在分析一个查询语句时，先首先执行常量表查询，然后把查询中涉及到该表的条件全部替换成常数，最后再分析其余表的查询成本，比方说这个查询语句：

```
SELECT * FROM table1 INNER JOIN table2
    ON table1.column1 = table2.column2
    WHERE table1.primary_key = 1;
```

很明显，这个查询可以使用主键和常量值的等值匹配来查询 table1 表，也就是在这个查询中 table1 表相当于 常量表 ，在分析对 table2 表的查询成本之前，就会执行对 table1 表的查询，并把查询中涉及 table1 表的条件都替换掉，也就是上边的语句会被转换成这样：

```
SELECT table1表记录的各个字段的常量值, table2.* FROM table1 INNER JOIN table2
    ON table1表column1列的常量值 = table2.column2;
```

14.2 外连接消除

我们前边说过， 内连接 的驱动表和被驱动表的位置可以相互转换，而 左（外）连接 和 右（外）连接 的驱动表和被驱动表是固定的。这就导致 内连接 可能通过优化表的连接顺序来降低整体的查询成本，而 外连接 却无法优化表的连接顺序。为了故事的顺利发展，我们还是把之前介绍连接原理时用过的 t1 和 t2 表请出来，为了防止大家早就忘掉了，我们再看一下这两个表的结构：

```
CREATE TABLE t1 (
    m1 int,
    n1 char(1)
) Engine=InnoDB, CHARSET=utf8;

CREATE TABLE t2 (
    m2 int,
    n2 char(1)
) Engine=InnoDB, CHARSET=utf8;
```

为了唤醒大家的记忆，我们再把这两个表中的数据给展示一下：

```
mysql> SELECT * FROM t1;
+------+------+
| m1   | n1   |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM t2;
+------+------+
| m2   | n2   |
+------+------+
|    2 | b    |
|    3 | c    |
|    4 | d    |
+------+------+
3 rows in set (0.00 sec)
```

我们之前说过，外连接和内连接的本质区别就是：对于外连接的驱动表的记录来说，如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录，那么该记录仍然会被加入到结果集中，对应的被驱动表记录的各个字段使用NULL值填充；而内连接的驱动表的记录如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录，那么该记录会被舍弃。查询效果就是这样：

```
mysql> SELECT * FROM t1 INNER JOIN t2 ON t1.m1 = t2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
|    1 | a    | NULL | NULL |
+------+------+------+------+
3 rows in set (0.00 sec)
```

对于上边例子中的（左）外连接来说，由于驱动表 t1 中 m1=1, n1='a' 的记录无法在被驱动表 t2 中找到符合 ON 子句条件 t1.m1 = t2.m2 的记录，所以就直接把这条记录加入到结果集，对应的 t2 表的 m2 和 n2 列的值都设置为 NULL 。

> 小贴士：
> 右（外）连接和左（外）连接其实只在驱动表的选取方式上是不同的，其余方面都是一样的，所以优化器会首先把右（外）连接查询转换成左（外）连接查询。我们后边就不再唠叨右（外）连接了。

我们知道 WHERE 子句的杀伤力比较大，凡是不符合WHERE子句中条件的记录都不会参与连接。只要我们在搜索条件中指定关于被驱动表相关列的值不为 NULL ，那么外连接中在被驱动表中找不到符合 ON 子句条件的驱动表记录也就被排除出最后的结果集了，也就是说：在这种情况下：外连接和内连接也就没有什么区别了！比方说这个查询：

```
mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.m2 WHERE t2.n2 IS NOT NULL;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.01 sec)
```

由于指定了被驱动表 t2 的 n2 列不允许为 NULL ，所以上边的 t1 和 t2 表的左（外）连接查询和内连接查询是一样一样的。当然，我们也可以不用显式的指定被驱动表的某个列 IS NOT NULL ，只要隐含的有这个意思就行了，比方说这样：

```
mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.m2 WHERE t2.m2 = 2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
+------+------+------+------+
1 row in set (0.00 sec)
```

在这个例子中，我们在 WHERE 子句中指定了被驱动表 t2 的 m2 列等于 2 ，也就相当于间接的指定了 m2 列不为 NULL 值，所以上边的这个左（外）连接查询其实和下边这个内连接查询是等价的：

```
mysql> SELECT * FROM t1 INNER JOIN t2 ON t1.m1 = t2.m2 WHERE t2.m2 = 2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
+------+------+------+------+
1 row in set (0.00 sec)
```

我们把这种在外连接查询中，指定的 WHERE 子句中包含被驱动表中的列不为 NULL 值的条件称之为 空值拒绝 （英文名： reject-NULL ）。在被驱动表的WHERE子句符合空值拒绝的条件后，外连接和内连接可以相互转换。这种转换带来的好处就是查询优化器可以通过评估表的不同连接顺序的成本，选出成本最低的那种连接顺序来执行查询。

14.3 子查询优化

我们的主题本来是唠叨 MySQL 查询优化器是如何处理子查询的，但是我还是有一万个担心好多同学连子查询的语法都没掌握全，所以我们就先唠叨唠叨什么是个子查询（当然不会面面俱到啦，只是说个大概哈），然后再唠叨关于子查询优化的事儿。

14.3.1 子查询语法

想必大家都是妈妈生下来的吧，连孙猴子都有妈妈——石头人。怀孕妈妈肚子里的那个东东就是她的孩子，类似的，在一个查询语句里的某个位置也可以有另一个查询语句，这个出现在某个查询语句的某个位置中的查询就被称为 子查询 （我们也可以称它为宝宝查询哈哈），那个充当“妈妈”角色的查询也被称之为 外层查询 。不像人们

怀孕时宝宝们都只在肚子里，子查询可以在一个外层查询的各种位置出现，比如：

- SELECT 子句中

  也就是我们平时说的查询列表中，比如这样：

```
mysql> SELECT (SELECT m1 FROM t1 LIMIT 1);
+-----------------------------+
| (SELECT m1 FROM t1 LIMIT 1) |
+-----------------------------+
|                           1 |
+-----------------------------+
1 row in set (0.00 sec)
```

  其中的 (SELECT m1 FROM t1 LIMIT 1) 就是我们唠叨的所谓的 子查询 。

- FROM 子句中

  比如：

```
SELECT m, n FROM (SELECT m2 + 1 AS m, n2 AS n FROM t2 WHERE m2 > 2) AS t;
+------+------+
| m    | n    |
+------+------+
|    4 | c    |
|    5 | d    |
+------+------+
2 rows in set (0.00 sec)
```

  这个例子中的子查询是： (SELECT m2 + 1 AS m, n2 AS n FROM t2 WHERE m2 > 2) ，很特别的地方是它出现在了 FROM 子句中。 FROM 子句里边儿不是存放我们要查询的表的名称么，这里放进来一个子查询是个什么鬼？其实这里我们可以把子查询的查询结果当作是一个表，子查询后边的 AS t 表明这个子查询的结果就相当于一个名称为 t 的表，这个名叫 t 的表的列就是子查询结果中的列，比如例子中表 t 就有两个列： m 列和 n 列。这个放在 FROM 子句中的子查询本质上相当于一个 表 ，但又和我们平常使用的表有点儿不一样，设计 MySQL 的大叔把这种由子查询结果集组成的表称之为 派生表 。

- WHERE 或 ON 子句中

  把子查询放在外层查询的 WHERE 子句或者 ON 子句中可能是我们最常用的一种使用子查询的方式了，比如这样：

```
mysql> SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2);
+------+------+
| m1   | n1   |
+------+------+
|    2 | b    |
|    3 | c    |
+------+------+
2 rows in set (0.00 sec)
```

  这个查询表明我们想要将 (SELECT m2 FROM t2) 这个子查询的结果作为外层查询的 IN 语句参数，整个查询语句的意思就是我们想找 t1 表中的某些记录，这些记录的 m1 列的值能在 t2 表的 m2 列找到匹配的值。

- ORDER BY 子句中

  虽然语法支持，但没啥子意义，不唠叨这种情况了。

- GROUP BY 子句中

同上~

14.3.1.1 按返回的结果集区分子查询

因为子查询本身也算是一个查询，所以可以按照它们返回的不同结果集类型而把这些子查询分为不同的类型：

- 标量子查询

  那些只返回一个单一值的子查询称之为 `标量子查询` ，比如这样：

  ```
  SELECT (SELECT m1 FROM t1 LIMIT 1);
  ```

  或者这样：

  ```
  SELECT * FROM t1 WHERE m1 = (SELECT MIN(m2) FROM t2);
  ```

  这两个查询语句中的子查询都返回一个单一的值，也就是一个 `标量` 。这些标量子查询可以作为一个单一值或者表达式的一部分出现在查询语句的各个地方。
- 行子查询

  顾名思义，就是返回一条记录的子查询，不过这条记录需要包含多个列（只包含一个列就成了标量子查询了）。比如这样：

  ```
  SELECT * FROM t1 WHERE (m1, n1) = (SELECT m2, n2 FROM t2 LIMIT 1);
  ```

  其中的 `(SELECT m2, n2 FROM t2 LIMIT 1)` 就是一个行子查询，整条语句的含义就是要从 `t1` 表中找一些记录，这些记录的 `m1` 和 `n2` 列分别等于子查询结果中的 `m2` 和 `n2` 列。
- 列子查询

  列子查询自然就是查询出一个列的数据喽，不过这个列的数据需要包含多条记录（只包含一条记录就成了标量子查询了）。比如这样：

  ```
  SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2);
  ```

  其中的 `(SELECT m2 FROM t2)` 就是一个列子查询，表明查询出 `t2` 表的 `m2` 列的值作为外层查询 `IN` 语句的参数。
- 表子查询

  顾名思义，就是子查询的结果既包含很多条记录，又包含很多个列，比如这样：

  ```
  SELECT * FROM t1 WHERE (m1, n1) IN (SELECT m2, n2 FROM t2);
  ```

  其中的 `(SELECT m2, n2 FROM t2)` 就是一个表子查询，这里需要和行子查询对比一下，行子查询中我们用了 `LIMIT 1` 来保证子查询的结果只有一条记录，表子查询中不需要这个限制。

14.3.1.2 按与外层查询关系来区分子查询

- 不相关子查询

  如果子查询可以单独运行出结果，而不依赖于外层查询的值，我们就可以把这个子查询称之为 `不相关子查询` 。我们前边介绍的那些子查询全部都可以看作不相关子查询，所以也就不举例子了哈。
- 相关子查询

  如果子查询的执行需要依赖于外层查询的值，我们就可以把这个子查询称之为 `相关子查询` 。比如：

  ```
  SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2 WHERE n1 = n2);
  ```

例子中的子查询是 (SELECT m2 FROM t2 WHERE n1 = n2) ，可是这个查询中有一个搜索条件是 n1 = n2 ，别忘了 n1 是表 t1 的列，也就是外层查询的列，也就是说子查询的执行需要依赖于外层查询的值，所以这个子查询就是一个 相关子查询 。

14.3.1.3 子查询在布尔表达式中的使用

你说写下边这样的子查询有啥意义：

```
SELECT (SELECT m1 FROM t1 LIMIT 1);
```

貌似没啥意义～ 我们平时用子查询最多的地方就是把它作为布尔表达式的一部分来作为搜索条件用在 WHERE 子句或者 ON 子句里。所以我们这里来总结一下子查询在布尔表达式中的使用场景。

- 使用 = 、 > 、 < 、 >= 、 <= 、 <> 、 != 、 <=> 作为布尔表达式的操作符

  这些操作符具体是啥意思就不用我多介绍了吧，如果你不知道的话，那我真的很佩服你是靠着啥勇气一口气看到这里的～ 为了方便，我们就把这些操作符称为 comparison_operator 吧，所以子查询组成的布尔表达式就长这样：

  ```
  操作数 comparison_operator (子查询)
  ```

  这里的 操作数 可以是某个列名，或者是一个常量，或者是一个更复杂的表达式，甚至可以是另一个子查询。但是需要注意的是，这里的子查询只能是标量子查询或者行子查询，也就是子查询的结果只能返回一个单一的值或者只能是一条记录。比如这样（标量子查询）：

  ```
  SELECT * FROM t1 WHERE m1 < (SELECT MIN(m2) FROM t2);
  ```

  或者这样（行子查询）：

  ```
  SELECT * FROM t1 WHERE (m1, n1) = (SELECT m2, n2 FROM t2 LIMIT 1);
  ```

- [NOT] IN/ANY/SOME/ALL子查询

  对于列子查询和表子查询来说，它们的结果集中包含很多条记录，这些记录相当于是一个集合，所以就不能单纯的和另外一个操作数使用 comparison_operator 来组成布尔表达式了， MySQL 通过下面的语法来支持某个操作数和一个集合组成一个布尔表达式：

  - IN 或者 NOT IN

    具体的语法形式如下：

    ```
    操作数 [NOT] IN (子查询)
    ```

    这个布尔表达式的意思是用来判断某个操作数在不在由子查询结果集组成的集合中，比如下边的查询的意思是找出 t1 表中的某些记录，这些记录存在于子查询的结果集中：

    ```
    SELECT * FROM t1 WHERE (m1, n2) IN (SELECT m2, n2 FROM t2);
    ```

  - ANY/SOME （ ANY 和 SOME 是同义词）

    具体的语法形式如下：

    ```
    操作数 comparison_operator ANY/SOME(子查询)
    ```

    这个布尔表达式的意思是只要子查询结果集中存在某个值和给定的操作数做 comparison_operator 比较结果为 TRUE ，那么整个表达式的结果就为 TRUE ，否则整个表达式的结果就为 FALSE 。比方说下边这个查询：

    ```
    SELECT * FROM t1 WHERE m1 > ANY(SELECT m2 FROM t2);
    ```

这个查询的意思就是对于 t1 表的某条记录的 m1 列的值来说，如果子查询 (SELECT m2 FROM t2) 的结果集中存在一个小于 m1 列的值，那么整个布尔表达式的值就是 TRUE ，否则为 FALSE ，也就是说只要 m1 列的值大于子查询结果集中最小的值，整个表达式的结果就是 TRUE ，所以上边的查询本质上等价于这个查询：

```
SELECT * FROM t1 WHERE m1 > (SELECT MIN(m2) FROM t2);
```

另外，=ANY相当于判断子查询结果集中是否存在某个值和给定的操作数相等，它的含义和IN是相同的。

- ALL

具体的语法形式如下：

```
操作数 comparison_operator ALL(子查询)
```

这个布尔表达式的意思是子查询结果集中所有的值和给定的操作数做 comparison_operator 比较结果为 TRUE ，那么整个表达式的结果就为 TRUE ，否则整个表达式的结果就为 FALSE 。比方说下边这个查询：

```
SELECT * FROM t1 WHERE m1 > ALL(SELECT m2 FROM t2);
```

这个查询的意思就是对于 t1 表的某条记录的 m1 列的值来说，如果子查询 (SELECT m2 FROM t2) 的结果集中的所有值都小于 m1 列的值，那么整个布尔表达式的值就是 TRUE ，否则为 FALSE ，也就是说只要 m1 列的值大于子查询结果集中最大的值，整个表达式的结果就是 TRUE ，所以上边的查询本质上等价于这个查询：

```
SELECT * FROM t1 WHERE m1 > (SELECT MAX(m2) FROM t2);
```

> 小贴士：
> 觉得ANY和ALL有点晕的同学多看两遍哈～

- EXISTS子查询

有的时候我们仅仅需要判断子查询的结果集中是否有记录，而不在乎它的记录具体是个啥，可以使用把 EXISTS 或者 NOT EXISTS 放在子查询语句前边，就像这样：

```
[NOT] EXISTS (子查询)
```

我们举一个例子啊：

```
SELECT * FROM t1 WHERE EXISTS (SELECT 1 FROM t2);
```

对于子查询 (SELECT 1 FROM t2) 来说，我们并不关心这个子查询最后到底查询出的结果是什么，所以查询列表里填 * 、某个列名，或者其他啥东西都无所谓，我们真正关心的是子查询的结果集中是否存在记录。也就是说只要 (SELECT 1 FROM t2) 这个查询中有记录，那么整个 EXISTS 表达式的结果就为 TRUE 。

14.3.1.4 子查询语法注意事项

- 子查询必须用小括号扩起来。

不扩起来的子查询是非法的，比如这样：

```
mysql> SELECT SELECT m1 FROM t1;

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'SELECT m1 FROM t1' at line 1
```

- 在 SELECT 子句中的子查询必须是标量子查询。

  如果子查询结果集中有多个列或者多个行，都不允许放在 SELECT 子句中，也就是查询列表中，比如这样就是非法的：

  ```
  mysql> SELECT (SELECT m1, n1 FROM t1);

  ERROR 1241 (21000): Operand should contain 1 column(s)
  ```

- 在想要得到标量子查询或者行子查询，但又不能保证子查询的结果集只有一条记录时，应该使用 LIMIT 1 语句来限制记录数量。
- 对于 [NOT] IN/ANY/SOME/ALL 子查询来说，子查询中不允许有 LIMIT 语句。

  比如这样是非法的：

  ```
  mysql> SELECT * FROM t1 WHERE m1 IN (SELECT * FROM t2 LIMIT 2);

  ERROR 1235 (42000): This version of MySQL doesn't yet support 'LIMIT & IN/ALL/ANY/SO
  ME subquery'
  ```

  为啥不合法？人家就这么规定的，不解释~ 可能以后的版本会支持吧。正因为 [NOT] IN/ANY/SOME/ALL 子查询不支持 LIMIT 语句，所以子查询中的这些语句也就是多余的了：

  - ORDER BY 子句

    子查询的结果其实就相当于一个集合，集合里的值排不排序一点儿都不重要，比如下边这个语句中的 ORDER BY 子句简直就是画蛇添足：

    ```
    SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2 ORDER BY m2);
    ```

  - DISTINCT 语句

    集合里的值去不去重也没啥意义，比如这样：

    ```
    SELECT * FROM t1 WHERE m1 IN (SELECT DISTINCT m2 FROM t2);
    ```

  - 没有聚集函数以及 HAVING 子句的 GROUP BY 子句。

    在没有聚集函数以及 HAVING 子句时， GROUP BY 子句就是个摆设，比如这样：

    ```
    SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2 GROUP BY m2);
    ```

    对于这些冗余的语句，查询优化器在一开始就把它们给干掉了。

- 不允许在一条语句中增删改某个表的记录时同时还对该表进行子查询。

  比方说这样：

  ```
  mysql> DELETE FROM t1 WHERE m1 < (SELECT MAX(m1) FROM t1);

  ERROR 1093 (HY000): You can't specify target table 't1' for update in FROM clause
  ```

14.3.2 子查询在MySQL中是怎么执行的

好了，关于子查询的基础语法我们用最快的速度温习了一遍，如果想了解更多语法细节，大家可以去查看一下 MySQL 的文档哈，现在我们就假设各位都懂了啥是个子查询了喔，接下来就要唠叨具体某种类型的子查询在 MySQL 中是怎么执行的了，想想就有点儿小激动呢~ 当然，为了故事的顺利发展，我们的例子也需要跟随形势鸟枪换炮，还是要祭出我们用了n遍的 single_table 表：

```
CREATE TABLE single_table (
    id INT NOT NULL AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1),
    UNIQUE KEY idx_key2 (key2),
    KEY idx_key3 (key3),
    KEY idx_key_part(key_part1, key_part2, key_part3)
) Engine=InnoDB CHARSET=utf8;
```

为了方便，我们假设有两个表 s1 、 s2 与这个 single_table 表的构造是相同的，而且这两个表里边儿有10000条记录，除id列外其余的列都插入随机值。下边正式开始我们的表演。

14.3.2.1 小白们眼中子查询的执行方式

在我还是一个单纯无知的少年时，觉得子查询的执行方式是这样的：

- 如果该子查询是不相关子查询，比如下边这个查询：

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT common_field FROM s2);
  ```

  我年少时觉得这个查询是的执行方式是这样的：
  - 先单独执行 (SELECT common_field FROM s2) 这个子查询。
  - 然后在将上一步子查询得到的结果当作外层查询的参数再执行外层查询 SELECT * FROM s1 WHERE key1 IN (...) 。
- 如果该子查询是相关子查询，比如下边这个查询：

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT common_field FROM s2 WHERE s1.key2 = s2.key2);
  ```

  这个查询中的子查询中出现了 s1.key2 = s2.key2 这样的条件，意味着该子查询的执行依赖着外层查询的值，所以我年少时觉得这个查询的执行方式是这样的：
  - 先从外层查询中获取一条记录，本例中也就是先从 s1 表中获取一条记录。
  - 然后从上一步骤中获取的那条记录中找出子查询中涉及到的值，本例中就是从 s1 表中获取的那条记录中找出 s1.key2 列的值，然后执行子查询。
  - 最后根据子查询的查询结果来检测外层查询 WHERE 子句的条件是否成立，如果成立，就把外层查询的那条记录加入到结果集，否则就丢弃。
  - 再次执行第一步，获取第二条外层查询中的记录，依次类推~

告诉我不只是我一个人是这样认为的，这样认为的同学请举起你们的双手~~~ 哇唔，还真不少~

其实设计 MySQL 的大叔想了一系列的办法来优化子查询的执行，大部分情况下这些优化措施其实挺有效的，但是保不齐有的时候马失前蹄，下边我们详细唠叨各种不同类型的子查询具体是怎么执行的。

> 小贴士：
> 我们下边即将唠叨的关于MySQL优化子查询的执行方式的事儿都是基于MySQL5.7这个版本的，以后版本可能有更新的优化策略！

14.3.2.2 标量子查询、行子查询的执行方式

我们经常在下边两个场景中使用到标量子查询或者行子查询：

- SELECT 子句中，我们前边说过的在查询列表中的子查询必须是标量子查询。
- 子查询使用 = 、 > 、 < 、 >= 、 <= 、 <> 、 != 、 <=> 等操作符和某个操作数组成一个布尔表达式，这样的子查询必须是标量子查询或者行子查询。

对于上述两种场景中的不相关标量子查询或者行子查询来说，它们的执行方式是简单的，比方说下边这个查询语句：

```
SELECT * FROM s1
    WHERE key1 = (SELECT common_field FROM s2 WHERE key3 = 'a' LIMIT 1);
```

它的执行方式和年少的我想的一样：

- 先单独执行 (SELECT common_field FROM s2 WHERE key3 = 'a' LIMIT 1) 这个子查询。
- 然后在将上一步子查询得到的结果当作外层查询的参数再执行外层查询 SELECT * FROM s1 WHERE key1 = ... 。

也就是说，对于包含不相关的标量子查询或者行子查询的查询语句来说，MySQL会分别独立的执行外层查询和子查询，就当作两个单表查询就好了。

对于相关的标量子查询或者行子查询来说，比如下边这个查询：

```
SELECT * FROM s1 WHERE
    key1 = (SELECT common_field FROM s2 WHERE s1.key3 = s2.key3 LIMIT 1);
```

事情也和年少的我想的一样，它的执行方式就是这样的：

- 先从外层查询中获取一条记录，本例中也就是先从 s1 表中获取一条记录。
- 然后从上一步骤中获取的那条记录中找出子查询中涉及到的值，本例中就是从 s1 表中获取的那条记录中找出 s1.key3 列的值，然后执行子查询。
- 最后根据子查询的查询结果来检测外层查询 WHERE 子句的条件是否成立，如果成立，就把外层查询的那条记录加入到结果集，否则就丢弃。
- 再次执行第一步，获取第二条外层查询中的记录，依次类推～

也就是说对于一开始唠叨的两种使用标量子查询以及行子查询的场景中， MySQL 优化器的执行方式并没有什么新鲜的。

14.3.2.3 IN子查询优化

物化表的提出

对于不相关的 IN 子查询，比如这样：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a');
```

我们最开始的感觉就是这种不相关的 IN 子查询和不相关的标量子查询或者行子查询是一样一样的，都是把外层查询和子查询当作两个独立的单表查询来对待，可是很遗憾的是设计 MySQL 的大叔为了优化 IN 子查询倾注了太多心血（毕竟 IN 子查询是我们日常生活中最常用的子查询类型），所以整个执行过程并不像我们想象的那么简单(>_<)。

其实说句老实话，对于不相关的 IN 子查询来说，如果子查询的结果集中的记录条数很少，那么把子查询和外层查询分别看成两个单独的单表查询效率还是蛮高的，但是如果单独执行子查询后的结果集太多的话，就会导致这些问题：

- 结果集太多，可能内存中都放不下～
- 对于外层查询来说，如果子查询的结果集太多，那就意味着 IN 子句中的参数特别多，这就导致：
  - 无法有效的使用索引，只能对外层查询进行全表扫描。
  - 在对外层查询执行全表扫描时，由于 IN 子句中的参数太多，这会导致检测一条记录是否符合和 IN 子句中的参数匹配花费的时间太长。

    比如说 IN 子句中的参数只有两个：

    ```
    SELECT * FROM tbl_name WHERE column IN (a, b);
    ```

    这样相当于需要对 tbl_name 表中的每条记录判断一下它的 column 列是否符合 column = a OR column = b 。在 IN 子句中的参数比较少时这并不是什么问题，如果 IN 子句中的参数比较多时，比如这样：

    ```
    SELECT * FROM tbl_name WHERE column IN (a, b, c ..., ...);
    ```

    那么这样每条记录需要判断一下它的 column 列是否符合 column = a OR column = b OR column = c OR ... ，这样性能耗费可就多了。

于是乎设计 MySQL 的大叔想了一个招：不直接将不相关子查询的结果集当作外层查询的参数，而是将该结果集写入一个临时表里。写入临时表的过程是这样的：

- 该临时表的列就是子查询结果集中的列。

- 写入临时表的记录会被去重。

  我们说 IN 语句是判断某个操作数在不在某个集合中，集合中的值重不重复对整个 IN 语句的结果并没有啥子关系，所以我们在将结果集写入临时表时对记录进行去重可以让临时表变得更小，更省地方～

  > 小贴士：
  > 临时表如何对记录进行去重？这不是小意思嘛，临时表也是个表，只要为表中记录的所有列建立主键或者唯一索引就好了嘛～

- 一般情况下子查询结果集不会大的离谱，所以会为它建立基于内存的使用 Memory 存储引擎的临时表，而且会为该表建立哈希索引。

  > 小贴士：
  > IN语句的本质就是判断某个操作数在不在某个集合里，如果集合中的数据建立了哈希索引，那么这个匹配的过程就是超级快的。
  > 有同学不知道哈希索引是什么？我这里就不展开了，自己上网找找吧，不会了再来问我～

  如果子查询的结果集非常大，超过了系统变量 tmp_table_size 或者 max_heap_table_size ，临时表会转而使用基于磁盘的存储引擎来保存结果集中的记录，索引类型也对应转变为 B+ 树索引。

设计 MySQL 的大叔把这个将子查询结果集中的记录保存到临时表的过程称之为 物化 （英文名：Materialize ）。为了方便起见，我们就把那个存储子查询结果集的临时表称之为 物化表 。正因为物化表中的记录都建立了索引（基于内存的物化表有哈希索引，基于磁盘的有B+树索引），通过索引执行 IN 语句判断某个操作数在不在子查询结果集中变得非常快，从而提升了子查询语句的性能。

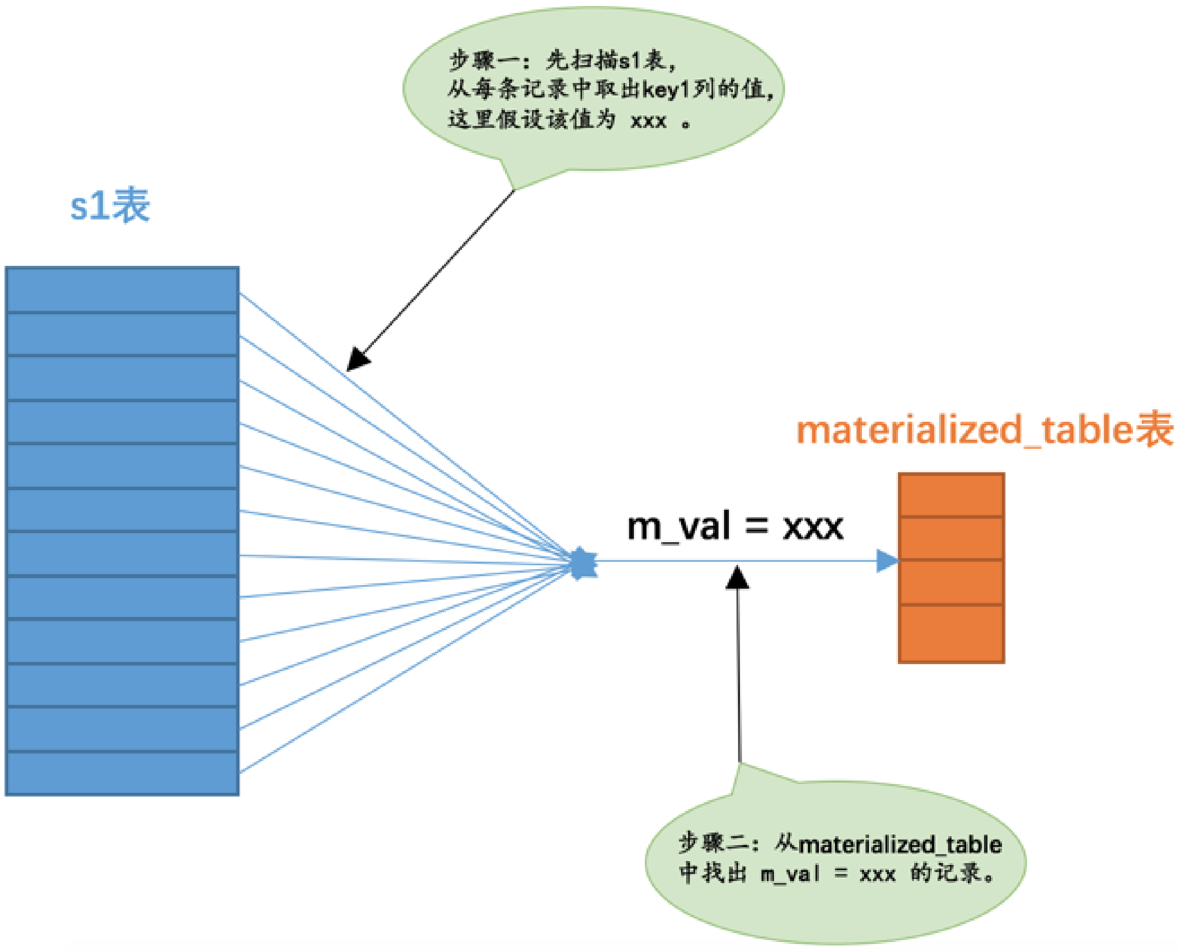
物化表转连接

事情到这就完了？我们还得重新审视一下最开始的那个查询语句：

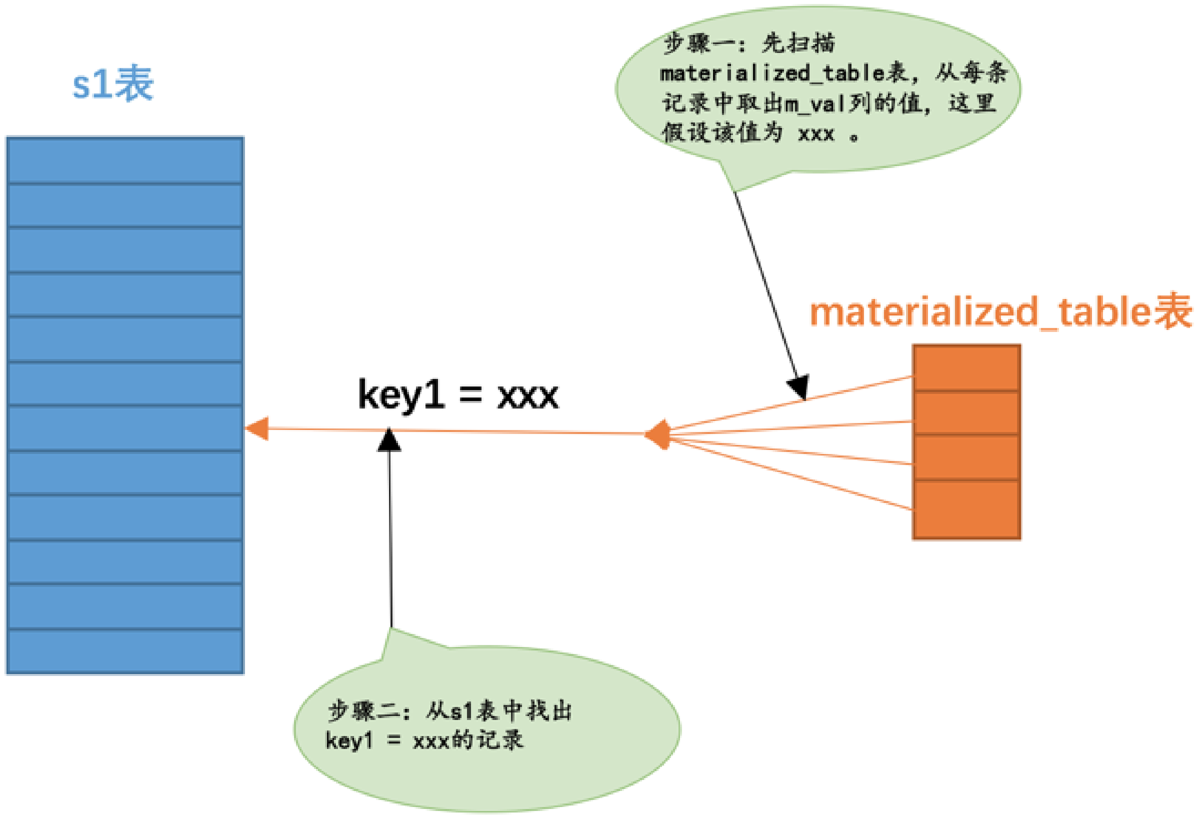
```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a');
```

当我们把子查询进行物化之后，假设子查询物化表的名称为 materialized_table ，该物化表存储的子查询结果集的列为 m_val ，那么这个查询其实可以从下边两种角度来看待：

- 从表 s1 的角度来看待，整个查询的意思其实是：对于 s1 表中的每条记录来说，如果该记录的 key1 列的值在子查询对应的物化表中，则该记录会被加入最终的结果集。画个图表示一下就是这样：



- 从子查询物化表的角度来看待，整个查询的意思其实是：对于子查询物化表的每个值来说，如果能在 s1 表中找到对应的 key1 列的值与该值相等的记录，那么就把这些记录加入到最终的结果集。画个图表示一下就是这样：



也就是说其实上边的查询就相当于表 s1 和子查询物化表 materialized_table 进行内连接：

```
SELECT s1.* FROM s1 INNER JOIN materialized_table ON key1 = m_val;
```

转化成内连接之后就有意思了，查询优化器可以评估不同连接顺序需要的成本是多少，选取成本最低的那种查询方式执行查询。我们分析一下上述查询中使用外层查询的表 s1 和物化表 materialized_table 进行内连接的成本都是由哪几部分组成的：

- 如果使用 s1 表作为驱动表的话，总查询成本由下边几个部分组成：
  - 物化子查询时需要的成本
  - 扫描 s1 表时的成本
  - s1表中的记录数量 × 通过 m_val = xxx 对 materialized_table 表进行单表访问的成本（我们前边说过物化表中的记录是不重复的，并且为物化表中的列建立了索引，所以这个步骤显然是非常快的）。
- 如果使用 materialized_table 表作为驱动表的话，总查询成本由下边几个部分组成：
  - 物化子查询时需要的成本
  - 扫描物化表时的成本
  - 物化表中的记录数量 × 通过 key1 = xxx 对 s1 表进行单表访问的成本（非常庆幸 key1 列上建立了索引，所以这个步骤是非常快的）。

MySQL 查询优化器会通过运算来选择上述成本更低的方案来执行查询。

将子查询转换为semi-join

虽然将子查询进行物化之后再执行查询都会有建立临时表的成本，但是不管怎么说，我们见识到了将子查询转换为连接的强大作用，设计 MySQL 的大叔继续开脑洞：能不能不进行物化操作直接把子查询转换为连接呢？让我们重新审视一下上边的查询语句：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a');
```

我们可以把这个查询理解成：对于 s1 表中的某条记录，如果我们能在 s2 表（准确的说是执行完 WHERE s2.key3 = 'a' 之后的结果集）中找到一条或多条记录，这些记录的 common_field 的值等于 s1 表记录的 key1 列的值，那么该条 s1 表的记录就会被加入到最终的结果集。这个过程其实和把 s1 和 s2 两个表连接起来的效果很像：

```
SELECT s1.* FROM s1 INNER JOIN s2
    ON s1.key1 = s2.common_field
    WHERE s2.key3 = 'a';
```

只不过我们不能保证对于 s1 表的某条记录来说，在 s2 表（准确的说是执行完 WHERE s2.key3 = 'a' 之后的结果集）中有多少条记录满足 s1.key1 = s2.common_field 这个条件，不过我们可以分三种情况讨论：

- 情况一：对于 s1 表的某条记录来说， s2 表中没有任何记录满足 s1.key1 = s2.common_field 这个条件，那么该记录自然也不会加入到最后的结果集。
- 情况二：对于 s1 表的某条记录来说， s2 表中有且只有记录满足 s1.key1 = s2.common_field 这个条件，那么该记录会被加入最终的结果集。
- 情况三：对于 s1 表的某条记录来说， s2 表中至少有2条记录满足 s1.key1 = s2.common_field 这个条件，那么该记录会被多次加入最终的结果集。

对于 s1 表的某条记录来说，由于我们只关心 s2 表中是否存在记录满足 s1.key1 = s2.common_field 这个条件，而不关心具体有多少条记录与之匹配，又因为有 情况三 的存在，我们上边所说的 IN 子查询和两表连接之间并不完全等价。但是将子查询转换为连接又真的可以充分发挥优化器的作用，所以设计 MySQL 的大叔在这里提出了一个新概念 --- 半连接 （英文名： semi-join ）。将 s1 表和 s2 表进行半连接的意思就是：对于 s1 表的某条记录来说，我们只关心在 s2 表中是否存在与之匹配的记录是否存在，而不关心具体有多少条记录与之匹配，最终的结果集中只保留 s1 表的记录。为了让大家有更直观的感受，我们假设MySQL内部是这么改写上边的子查询的：

```
SELECT s1.* FROM s1 SEMI JOIN s2
    ON s1.key1 = s2.common_field
    WHERE key3 = 'a';
```

> 小贴士：
> semi-join只是在MySQL内部采用的一种执行子查询的方式，MySQL并没有提供面向用户的semi-join语法，所以我们不需要，也不能尝试把上边这个语句放到黑框框里运行，我只是想说明一下上边的子查询在MySQL内部会被转换为类似上边语句的半连接～

概念是有了，怎么实现这种所谓的 半连接 呢？设计 MySQL 的大叔准备了好几种办法。

- Table pullout （子查询中的表上拉）

  当子查询的查询列表处只有主键或者唯一索引列时，可以直接把子查询中的表 上拉 到外层查询的 FROM 子句中，并把子查询中的搜索条件合并到外层查询的搜索条件中，比如这个

  ```
  SELECT * FROM s1
      WHERE key2 IN (SELECT key2 FROM s2 WHERE key3 = 'a');
  ```

  由于 key2 列是 s2 表的唯一二级索引列，所以我们可以直接把 s2 表上拉到外层查询的 FROM 子句中，并且把子查询中的搜索条件合并到外层查询的搜索条件中，上拉之后的查询就是这样的：

  ```
  SELECT s1.* FROM s1 INNER JOIN s2
      ON s1.key2 = s2.key2
      WHERE s2.key3 = 'a';
  ```

  为啥当子查询的查询列表处只有主键或者唯一索引列时，就可以直接将子查询转换为连接查询呢？哎呀，主键或者唯一索引列中的数据本身就是不重复的嘛！所以对于同一条 s1 表中的记录，你不可能找到两条以上的符合 s1.key2 = s2.key2 的记录呀～
- DuplicateWeedout execution strategy （重复值消除）

对于这个查询来说：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a');
```

转换为半连接查询后， s1 表中的某条记录可能在 s2 表中有多条匹配的记录，所以该条记录可能多次被添加到最后的结果集中，为了消除重复，我们可以建立一个临时表，比方说这个临时表长这样：

```
CREATE TABLE tmp (
    id PRIMARY KEY
);
```

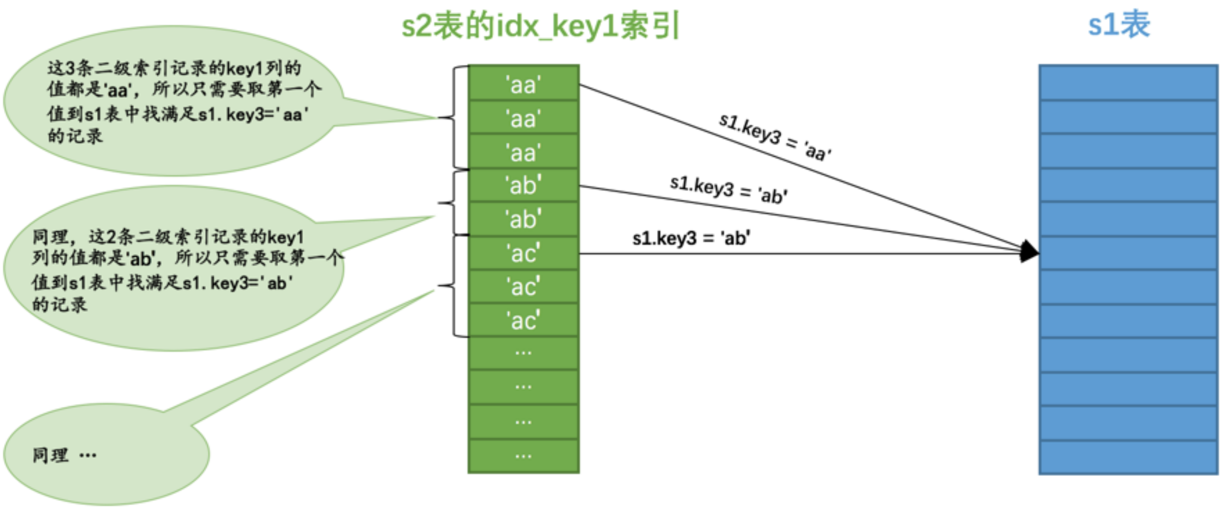
这样在执行连接查询的过程中，每当某条 s1 表中的记录要加入结果集时，就首先把这条记录的 id 值加入到这个临时表里，如果添加成功，说明之前这条 s1 表中的记录并没有加入最终的结果集，现在把该记录添加到最终的结果集；如果添加失败，说明这条之前这条 s1 表中的记录已经加入过最终的结果集，这里直接把它丢弃就好了，这种使用临时表消除 semi-join 结果集中的重复值的方式称之为 DuplicateWeedout 。

- LooseScan execution strategy （松散索引扫描）

大家看这个查询：

```
SELECT * FROM s1
    WHERE key3 IN (SELECT key1 FROM s2 WHERE key1 > 'a' AND key1 < 'b');
```

在子查询中，对于 s2 表的访问可以使用到 key1 列的索引，而恰好子查询的查询列表处就是 key1 列，这样在将该查询转换为半连接查询后，如果将 s2 作为驱动表执行查询的话，那么执行过程就是这样：



如图所示，在 s2 表的 idx_key1 索引中，值为 'aa' 的二级索引记录一共有3条，那么只需要取第一条的值到 s1 表中查找 s1.key3 = 'aa' 的记录，如果能在 s1 表中找到对应的记录，那么就把对应的记录加入到结果集。依此类推，其他值相同的二级索引记录，也只需要取第一条记录的值到 s1 表中找匹配的记录，这种虽然是扫描索引，但只取值相同的记录的第一条去做匹配操作的方式称之为 松散索引扫描 。

- Semi-join Materialization execution strategy

我们之前介绍的先把外层查询的 IN 子句中的不相关子查询进行物化，然后再进行外层查询的表和物化表的连接本质上也算是一种 semi-join ，只不过由于物化表中没有重复的记录，所以可以直接将子查询转为连接查询。

- FirstMatch execution strategy （首次匹配）

FirstMatch 是一种最原始的半连接执行方式，跟我们年少时认为的相关子查询的执行方式是一样一样的，就是说先取一条外层查询的中的记录，然后到子查询的表中寻找符合匹配条件的记录，如果能找到一条，则将该外层查询的记录放入最终的结果集并且停止查找更多匹配的记录，如果找不到则把该外层查询的记录丢弃掉；然后再开始取下一条外层查询中的记录，重复上边这个过程。

对于某些使用 IN 语句的相关子查询，比方这个查询：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE s1.key3 = s2.key3);
```

它也可以很方便的转为半连接，转换后的语句类似这样：

```
SELECT s1.* FROM s1 SEMI JOIN s2
    ON s1.key1 = s2.common_field AND s1.key3 = s2.key3;
```

然后就可以使用我们上边介绍过的 DuplicateWeedout 、 LooseScan 、 FirstMatch 等半连接执行策略来执行查询，当然，如果子查询的查询列表处只有主键或者唯一二级索引列，还可以直接使用 table pullout 的策略来执行查询，但是需要大家注意的是，由于相关子查询并不是一个独立的查询，所以不能转换为物化表来执行查询。

semi-join的适用条件

当然，并不是所有包含 IN 子查询的查询语句都可以转换为 semi-join ，只有形如这样的查询才可以被转换为 semi-join ：

```
SELECT ... FROM outer_tables
    WHERE expr IN (SELECT ... FROM inner_tables ...) AND ...
```

或者这样的形式也可以：

```
SELECT ... FROM outer_tables
    WHERE (oe1, oe2, ...) IN (SELECT ie1, ie2, ... FROM inner_tables ...) AND ...
```

用文字总结一下，只有符合下边这些条件的子查询才可以被转换为 semi-join ：

- 该子查询必须是和 IN 语句组成的布尔表达式，并且在外层查询的 WHERE 或者 ON 子句中出现。
- 外层查询也可以有其他的搜索条件，只不过和 IN 子查询的搜索条件必须使用 AND 连接起来。
- 该子查询必须是一个单一的查询，不能是由若干查询由 UNION 连接起来的形式。
- 该子查询不能包含 GROUP BY 或者 HAVING 语句或者聚集函数。
- ... 还有一些条件比较少见，就不唠叨啦～

不适用于semi-join的情况

对于一些不能将子查询转位 semi-join 的情况，典型的比如下边这几种：

- 外层查询的WHERE条件中有其他搜索条件与IN子查询组成的布尔表达式使用 OR 连接起来

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a')
          OR key2 > 100;
  ```

- 使用 NOT IN 而不是 IN 的情况

  ```
  SELECT * FROM s1
      WHERE key1 NOT IN (SELECT common_field FROM s2 WHERE key3 = 'a')
  ```

- 在 SELECT 子句中的IN子查询的情况

```
SELECT key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a') FROM s1 ;
```

- 子查询中包含 GROUP BY 、 HAVING 或者聚集函数的情况

```
SELECT * FROM s1
    WHERE key2 IN (SELECT COUNT(*) FROM s2 GROUP BY key1);
```

- 子查询中包含 UNION 的情况

```
SELECT * FROM s1 WHERE key1 IN (
    SELECT common_field FROM s2 WHERE key3 = 'a'
    UNION
    SELECT common_field FROM s2 WHERE key3 = 'b'
);
```

MySQL 仍然留了两手绝活来优化不能转为 semi-join 查询的子查询，那就是：

- 对于不相关子查询来说，可以尝试把它们物化之后再参与查询

  比如我们上边提到的这个查询：

  ```
  SELECT * FROM s1
      WHERE key1 NOT IN (SELECT common_field FROM s2 WHERE key3 = 'a')
  ```

  先将子查询物化，然后再判断 key1 是否在物化表的结果集中可以加快查询执行的速度。

  > 小贴士：
  > 请注意这里将子查询物化之后不能转为和外层查询的表的连接，只能是先扫描s1表，然后对s1表的某条记录来说，判断该记录的key1值在不在物化表中。

- 不管子查询是相关的还是不相关的，都可以把 IN 子查询尝试专为 EXISTS 子查询

  其实对于任意一个IN子查询来说，都可以被转为 EXISTS 子查询，通用的例子如下：

  ```
  outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
  ```

  可以被转换为：

  ```
  EXISTS (SELECT inner_expr FROM ... WHERE subquery_where AND outer_expr=inner_expr)
  ```

  当然这个过程中有一些特殊情况，比如在 outer_expr 或者 inner_expr 值为 NULL 的情况下就比较特殊。因为有 NULL 值作为操作数的表达式结果往往是 NULL ，比方说：

```
mysql> SELECT NULL IN (1, 2, 3);
+-------------------+
| NULL IN (1, 2, 3) |
+-------------------+
|              NULL |
+-------------------+
1 row in set (0.00 sec)

mysql> SELECT 1 IN (1, 2, 3);
+----------------+
| 1 IN (1, 2, 3) |
+----------------+
|              1 |
+----------------+
1 row in set (0.00 sec)

mysql> SELECT NULL IN (NULL);
+----------------+
| NULL IN (NULL) |
+----------------+
|           NULL |
+----------------+
1 row in set (0.00 sec)
```

而 EXISTS 子查询的结果肯定是 TRUE 或者 FASLE ：

```
mysql> SELECT EXISTS (SELECT 1 FROM s1 WHERE NULL = 1);
+------------------------------------------+
| EXISTS (SELECT 1 FROM s1 WHERE NULL = 1) |
+------------------------------------------+
|                                        0 |
+------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT EXISTS (SELECT 1 FROM s1 WHERE 1 = NULL);
+------------------------------------------+
| EXISTS (SELECT 1 FROM s1 WHERE 1 = NULL) |
+------------------------------------------+
|                                        0 |
+------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT EXISTS (SELECT 1 FROM s1 WHERE NULL = NULL);
+---------------------------------------------+
| EXISTS (SELECT 1 FROM s1 WHERE NULL = NULL) |
+---------------------------------------------+
|                                           0 |
+---------------------------------------------+
1 row in set (0.00 sec)
```

但是幸运的是，我们大部分使用 IN 子查询的场景是把它放在 WHERE 或者 ON 子句中，而 WHERE 或者 ON 子句是不区分 NULL 和 FALSE 的，比方说：

```
mysql> SELECT 1 FROM s1 WHERE NULL;
Empty set (0.00 sec)

mysql> SELECT 1 FROM s1 WHERE FALSE;
Empty set (0.00 sec)
```

所以只要我们的 `IN` 子查询是放在 `WHERE` 或者 `ON` 子句中的，那么 `IN -> EXISTS` 的转换就是没问题的。说了这么多，为啥要转换呢？这是因为不转换的话可能用不到索引，比方说下边这个查询：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT key3 FROM s2 where s1.common_field = s2.common_field)
        OR key2 > 1000;
```

这个查询中的子查询是一个相关子查询，而且子查询执行的时候不能使用到索引，但是将它转为 `EXISTS` 子查询后却可以使用到索引：

```
SELECT * FROM s1
    WHERE EXISTS (SELECT 1 FROM s2 where s1.common_field = s2.common_field AND s2.ke
y3 = s1.key1)
        OR key2 > 1000;
```

转为 `EXISTS` 子查询时便可以使用到 `s2` 表的 `idx_key3` 索引了。

需要注意的是，如果 `IN` 子查询不满足转换为 `semi-join` 的条件，又不能转换为物化表或者转换为物化表的成本太大，那么它就会被转换为 `EXISTS` 查询。

```
小贴士：
在MySQL5.5以及之前的版本没有引进semi-join和物化的方式优化子查询时，优化器都会把IN子
查询转换为EXISTS子查询，好多同学就惊呼我明明写的是一个不相关子查询，为啥要按照执行相关
子查询的方式来执行呢？所以当时好多声音都是建议大家把子查询转为连接，不过随着MySQL的发
展，最近的版本中引入了非常多的子查询优化策略，大家可以稍微放心的使用子查询了，内部的转
换工作优化器会为大家自动实现。
```

小结一下

- 如果 `IN` 子查询符合转换为 `semi-join` 的条件，查询优化器会优先把该子查询为 `semi-join` ，然后再考虑下边5种执行半连接的策略中哪个成本最低：
  - Table pullout
  - DuplicateWeedout
  - LooseScan
  - Materialization
  - FirstMatch

  选择成本最低的那种执行策略来执行子查询。
- 如果 `IN` 子查询不符合转换为 `semi-join` 的条件，那么查询优化器会从下边两种策略中找出一种成本更低的方式执行子查询：
  - 先将子查询物化之后再执行查询
  - 执行 `IN to EXISTS` 转换。

14.3.2.4 ANY/ALL子查询优化

如果ANY/ALL子查询是不相关子查询的话，它们在很多场合都能转换成我们熟悉的方式去执行，比方说：

原始表达式	转换为

原始表达式	转换为
< ANY (SELECT inner_expr ...)	< (SELECT MAX(inner_expr) ...)
> ANY (SELECT inner_expr ...)	> (SELECT MIN(inner_expr) ...)
< ALL (SELECT inner_expr ...)	< (SELECT MIN(inner_expr) ...)
> ALL (SELECT inner_expr ...)	> (SELECT MAX(inner_expr) ...)

14.3.2.5 [NOT] EXISTS子查询的执行

如果 [NOT] EXISTS 子查询是不相关子查询，可以先执行子查询，得出该 [NOT] EXISTS 子查询的结果是 TRUE 还是 FALSE ，并重写原先的查询语句，比如对这个查询来说：

```
SELECT * FROM s1
    WHERE EXISTS (SELECT 1 FROM s2 WHERE key1 = 'a')
        OR key2 > 100;
```

因为这个语句里的子查询是不相关子查询，所以优化器会首先执行该子查询，假设该EXISTS子查询的结果为 TRUE ，那么接着优化器会重写查询为：

```
SELECT * FROM s1
    WHERE TRUE OR key2 > 100;
```

进一步简化后就变成了：

```
SELECT * FROM s1
    WHERE TRUE;
```

对于相关的 [NOT] EXISTS 子查询来说，比如这个查询：

```
SELECT * FROM s1
    WHERE EXISTS (SELECT 1 FROM s2 WHERE s1.common_field = s2.common_field);
```

很不幸，这个查询只能按照我们年少时的那种执行相关子查询的方式来执行。不过如果 [NOT] EXISTS 子查询中如果可以使用索引的话，那查询速度也会加快不少，比如：

```
SELECT * FROM s1
    WHERE EXISTS (SELECT 1 FROM s2 WHERE s1.common_field = s2.key1);
```

上边这个 EXISTS 子查询中可以使用 idx_key1 来加快查询速度。

14.3.2.6 对于派生表的优化

我们前边说过把子查询放在外层查询的 FROM 子句后，那么这个子查询的结果相当于一个 派生表 ，比如下边这个查询：

```
SELECT * FROM  (
        SELECT id AS d_id,  key3 AS d_key3 FROM s2 WHERE key1 = 'a'
    ) AS derived_s1 WHERE d_key3 = 'a';
```

子查询 (SELECT id AS d_id, key3 AS d_key3 FROM s2 WHERE key1 = 'a') 的结果就相当于一个派生表，这个表的名称是 derived_s1 ，该表有两个列，分别是 d_id 和 d_key3 。

对于含有 派生表 的查询， MySQL 提供了两种执行策略：

- 最容易想到的就是把派生表物化。

我们可以将派生表的结果集写到一个内部的临时表中，然后就把这个物化表当作普通表一样参与查询。当然，在对派生表进行物化时，设计 MySQL 的大叔使用了一种称为 延迟物化 的策略，也就是在查询中真正使用到派生表时才回去尝试物化派生表，而不是还没开始执行查询呢就把派生表物化掉。比方说对于下边这个含有派生表的查询来说：

```
SELECT * FROM (
        SELECT * FROM s1 WHERE key1 = 'a'
    ) AS derived_s1 INNER JOIN s2
    ON derived_s1.key1 = s2.key1
    WHERE s2.key2 = 1;
```

如果采用物化派生表的方式来执行这个查询的话，那么执行时首先会到 s1 表中找出满足 s1.key2 = 1 的记录，如果压根儿找不到，说明参与连接的 s1 表记录就是空的，所以整个查询的结果集就是空的，所以也就没有必要去物化查询中的派生表了。

- 将派生表和外层的表合并，也就是将查询重写为没有派生表的形式

  我们来看这个贼简单的包含派生表的查询：

  ```
  SELECT * FROM (SELECT * FROM s1 WHERE key1 = 'a') AS derived_s1;
  ```

  这个查询本质上就是想查看 s1 表中满足 key1 = 'a' 条件的的全部记录，所以和下边这个语句是等价的：

  ```
  SELECT * FROM s1 WHERE key1 = 'a';
  ```

  对于一些稍微复杂的包含派生表的语句，比如我们上边提到的那个：

  ```
  SELECT * FROM (
          SELECT * FROM s1 WHERE key1 = 'a'
      ) AS derived_s1 INNER JOIN s2
      ON derived_s1.key1 = s2.key1
      WHERE s2.key2 = 1;
  ```

  我们可以将派生表与外层查询的表合并，然后将派生表中的搜索条件放到外层查询的搜索条件中，就像这样：

  ```
  SELECT * FROM s1 INNER JOIN s2
      ON s1.key1 = s2.key1
      WHERE s1.key1 = 'a' AND s2.key2 = 1;
  ```

  这样通过将外层查询和派生表合并的方式成功的消除了派生表，也就意味着我们没必要再付出创建和访问临时表的成本了。可是并不是所有带有派生表的查询都能被成功的和外层查询合并，当派生表中有这些语句就不可以和外层查询合并：

  - 聚集函数，比如MAX()、MIN()、SUM()啥的
  - DISTINCT
  - GROUP BY
  - HAVING
  - LIMIT
  - UNION 或者 UNION ALL
  - 派生表对应的子查询的 SELECT 子句中含有另一个子查询
  - ... 还有些不常用的情况就不多说了哈~

所以 MySQL 在执行带有派生表的时候，优先尝试把派生表和外层查询合并掉，如果不行的话，再把派生表物化掉执行查询。

15 第15章 查询优化的百科全书-Explain详解（上）

标签： MySQL 是怎样运行的

一条查询语句在经过 MySQL 查询优化器的各种基于成本和规则的优化会后生成一个所谓的 执行计划 ，这个执行计划展示了接下来具体执行查询的方式，比如多表连接的顺序是什么，对于每个表采用什么访问方法来具体执行查询等等。设计 MySQL 的大叔贴心的为我们提供了 EXPLAIN 语句来帮助我们查看某个查询语句的具体执行计划，本章的内容就是为了帮助大家看懂 EXPLAIN 语句的各个输出项都是干嘛使的，从而可以有针对性的提升我们查询语句的性能。

如果我们想看看某个查询的执行计划的话，可以在具体的查询语句前边加一个 EXPLAIN ，就像这样：

```
mysql> EXPLAIN SELECT 1;
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | r
ows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+----------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | N
ULL |     NULL | No tables used |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+----------------+
1 row in set, 1 warning (0.01 sec)
```

然后这输出的一大坨东西就是所谓的 执行计划 ，我的任务就是带领大家看懂这一大坨东西里边的每个列都是干啥用的，以及在这个 执行计划 的辅助下，我们应该怎样改进自己的查询语句以使查询执行起来更高效。其实除了以 SELECT 开头的查询语句，其余的 DELETE 、 INSERT 、 REPLACE 以及 UPDATE 语句前边都可以加上 EXPLAIN 这个词儿，用来查看这些语句的执行计划，不过我们这里对 SELECT 语句更感兴趣，所以后边只会以 SELECT 语句为例来描述 EXPLAIN 语句的用法。为了让大家先有一个感性的认识，我们把 EXPLAIN 语句输出的各个列的作用先大致罗列一下：

列名	描述
id	在一个大的查询语句中每个 SELECT 关键字都对应一个唯一的 id
select_type	SELECT 关键字对应的那个查询的类型
table	表名
partitions	匹配的分区信息
type	针对单表的访问方法
possible_keys	可能用到的索引
key	实际上使用的索引
key_len	实际使用到的索引长度
ref	当使用索引列等值查询时，与索引列进行等值匹配的对象信息
rows	预估的需要读取的记录条数
filtered	某个表经过搜索条件过滤后剩余记录条数的百分比
Extra	一些额外的信息

需要注意的是，大家如果看不懂上边输出列含义，那是正常的，千万不要纠结～。我在这里把它们都列出来只是为了描述一个轮廓，让大家有一个大致的印象，下边会细细道来，等会儿说完了不信你不会～ 为了故事的顺利发展，我们还是要请出我们前边已经用了n遍的 single_table 表，为了防止大家忘了，再把它的结构描述一遍：

```
CREATE TABLE single_table (
    id INT NOT NULL AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1),
    UNIQUE KEY idx_key2 (key2),
    KEY idx_key3 (key3),
    KEY idx_key_part(key_part1, key_part2, key_part3)
) Engine=InnoDB CHARSET=utf8;
```

我们仍然假设有两个和 single_table 表构造一模一样的 s1 、 s2 表，而且这两个表里边儿有10000条记录，除id列外其余的列都插入随机值。为了让大家有比较好的阅读体验，我们下边并不准备严格按照 EXPLAIN 输出列的顺序来介绍这些列分别是干嘛的，大家注意一下就好了。

15.1 执行计划输出中各列详解

15.1.1 table

不论我们的查询语句有多复杂，里边儿包含了多少个表，到最后也是需要对每个表进行单表访问的，所以设计 MySQL 的大叔规定EXPLAIN语句输出的每条记录都对应着某个单表的访问方法，该条记录的table列代表着该表的表名。所以我们看一条比较简单的查询语句：

```
mysql> EXPLAIN SELECT * FROM s1;
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | r
ows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

这个查询语句只涉及对 s1 表的单表查询，所以 EXPLAIN 输出中只有一条记录，其中的 table 列的值是 s1 ，表明这条记录是用来说明对 s1 表的单表访问方法的。

下边我们看一下一个连接查询的执行计划：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+---------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | r
ows | filtered | Extra                                 |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+---------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
688 |   100.00 | NULL                                  |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
954 |   100.00 | Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+---------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

可以看到这个连接查询的执行计划中有两条记录，这两条记录的 table 列分别是 s1 和 s2 ，这两条记录用来分别说明对 s1 表和 s2 表的访问方法是什么。

15.1.2 id

我们知道我们写的查询语句一般都以 SELECT 关键字开头，比较简单的查询语句里只有一个 SELECT 关键字，比如下边这个查询语句：

```
SELECT * FROM s1 WHERE key1 = 'a';
```

稍微复杂一点的连接查询中也只有一个 SELECT 关键字，比如：

```
SELECT * FROM s1 INNER JOIN s2
    ON s1.key1 = s2.key1
    WHERE s1.common_field = 'a';
```

但是下边两种情况下在一条查询语句中会出现多个 SELECT 关键字：

- 查询中包含子查询的情况

  比如下边这个查询语句中就包含2个 SELECT 关键字：

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT * FROM s2);
  ```

- 查询中包含 UNION 语句的情况

  比如下边这个查询语句中也包含2个 SELECT 关键字：

  ```
  SELECT * FROM s1  UNION SELECT * FROM s2;
  ```

查询语句中每出现一个 SELECT 关键字，设计 MySQL 的大叔就会为它分配一个唯一的 id 值。这个 id 值就是 EXPLAIN 语句的第一个列，比如下边这个查询中只有一个 SELECT 关键字，所以 EXPLAIN 的结果中也就只有一条 id 列为 1 的记录：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref
| rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | cons
t |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

对于连接查询来说，一个 SELECT 关键字后边的 FROM 子句中可以跟随多个表，所以在连接查询的执行计划中，**每个表都会对应一条记录，但是这些记录的id值都是相同的**，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+---------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | r
ows | filtered | Extra                                 |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+---------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
688 |   100.00 | NULL                                  |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
954 |   100.00 | Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+---------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

可以看到，上述连接查询中参与连接的 s1 和 s2 表分别对应一条记录，但是这两条记录对应的 id 值都是 1 。这里需要大家记住的是，**在连接查询的执行计划中，每个表都会对应一条记录，这些记录的id列的值是相同的，出现在前边的表表示驱动表，出现在后边的表表示被驱动表**。所以从上边的 EXPLAIN 输出中我们可以看出，查询优化器准备让 s1 表作为驱动表，让 s2 表作为被驱动表来执行查询。

对于包含子查询的查询语句来说，就可能涉及多个 SELECT 关键字，所以在包含子查询的查询语句的执行计划中，每个 SELECT 关键字都会对应一个唯一的 id 值，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref
| rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NUL
L | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NUL
L | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+-------------+
2 rows in set, 1 warning (0.02 sec)
```

从输出结果中我们可以看到， s1 表在外层查询中，外层查询有一个独立的 SELECT 关键字，所以第一条记录的 id 值就是 1 ， s2 表在子查询中，子查询有一个独立的 SELECT 关键字，所以第二条记录的 id 值就是 2 。

但是这里大家需要特别注意，查询优化器可能对涉及子查询的查询语句进行重写，从而转换为连接查询。所以如果我们想知道查询优化器对某个包含子查询的语句是否进行了重写，直接查看执行计划就好了，比如说：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key3 FROM s2 WHERE common_field = 'a');
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+------------------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra                        |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+------------------------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key3      | NULL     | NULL    | NULL              | 9954 |    10.00 | Using where; Start temporary |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key3 |    1 |   100.00 | End temporary                |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到，虽然我们的查询语句是一个子查询，但是执行计划中 s1 和 s2 表对应的记录的 id 值全部是 1 ，这就表明了查询优化器将子查询转换为了连接查询。

对于包含 UNION 子句的查询语句来说，每个 SELECT 关键字对应一个 id 值也是没错的，不过还是有点儿特别的东西，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1  UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL     | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
3 rows in set, 1 warning (0.00 sec)
```

这个语句的执行计划的第三条记录是个什么鬼？为毛 id 值是 NULL ，而且 table 列长的也怪怪的？大家别忘了 UNION 子句是干嘛用的，它会把多个查询的结果集合并起来并对结果集中的记录进行去重，怎么去重呢？ MySQL 使用的是内部的临时表。正如上边的查询计划中所示， UNION 子句是为了把 id 为 1 的查询和 id 为 2 的查询的结果集合并起来并去重，所以在内部创建了一个名为 <union1, 2> 的临时表（就是执行计划第三条记录的 table 列的名称）， id 为 NULL 表明这个临时表是为了合并两个查询的结果集而创建的。

跟 UNION 对比起来， UNION ALL 就不需要为最终的结果集进行去重，它只是单纯的把多个查询的结果集中的记录合并成一个并返回给用户，所以也就不需要使用临时表。所以在包含 UNION ALL 子句的查询的执行计划中，就没有那个 id 为 NULL 的记录，如下所示：

```
mysql> EXPLAIN SELECT * FROM s1  UNION ALL SELECT * FROM s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | r
ows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+-------+
|  1 | PRIMARY     | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
688 |   100.00 | NULL  |
|  2 | UNION       | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9
954 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+--
----+----------+-------+
2 rows in set, 1 warning (0.01 sec)
```

15.1.3 select_type

通过上边的内容我们知道，一条大的查询语句里边可以包含若干个 SELECT 关键字，每个 SELECT 关键字代表着一个小的查询语句，而每个 SELECT 关键字的 FROM 子句中都可以包含若干张表（这些表用来做连接查询），每一张表都对应着执行计划输出中的一条记录，对于在同一个 SELECT 关键字中的表来说，它们的 id 值是相同的。

设计 MySQL 的大叔为每一个 SELECT 关键字代表的小查询都定义了一个称之为 select_type 的属性，意思是我们只要知道了某个小查询的 select_type 属性，就知道了这个小查询在整个大查询中扮演了一个什么角色，口说无凭，我们还是先来见识见识这个 select_type 都能取哪些值（为了精确起见，我们直接使用文档中的英文做简要描述，随后会进行详细解释的）：

名称	描述
SIMPLE	Simple SELECT (not using UNION or subqueries)
PRIMARY	Outermost SELECT
UNION	Second or later SELECT statement in a UNION
UNION RESULT	Result of a UNION
SUBQUERY	First SELECT in subquery
DEPENDENT SUBQUERY	First SELECT in subquery, dependent on outer query
DEPENDENT UNION	Second or later SELECT statement in a UNION, dependent on outer query
DERIVED	Derived table
MATERIALIZED	Materialized subquery
UNCACHEABLE SUBQUERY	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
UNCACHEABLE UNION	The second or later select in a UNION that belongs to an uncacheable subquery (see UNCACHEABLE SUBQUERY)

英文描述太简单，不知道说了啥？来详细瞅瞅里边儿的每个值都是干啥吃的：

- SIMPLE

  查询语句中不包含 UNION 或者子查询的查询都算作是 SIMPLE 类型，比方说下边这个单表查询的 select_type 的值就是 SIMPLE ：

```
mysql> EXPLAIN SELECT * FROM s1;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

当然，连接查询也算是 SIMPLE 类型，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+---------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra                                 |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+---------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | NULL                                  |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9954 |   100.00 | Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+---------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

- PRIMARY

对于包含 UNION 、 UNION ALL 或者子查询的大查询来说，它是由几个小查询组成的，其中最左边的那个查询的 select_type 值就是 PRIMARY ，比方说：

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+-------
--+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_le
n | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+-------
--+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL
| NULL | 9688 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL
| NULL | 9954 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL
| NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+-------
--+------+------+----------+-----------------+
3 rows in set, 1 warning (0.00 sec)
```

从结果中可以看到，最左边的小查询 SELECT * FROM s1 对应的是执行计划中的第一条记录，它的 select_type 值就是 PRIMARY 。

- UNION

  对于包含 UNION 或者 UNION ALL 的大查询来说，它是由几个小查询组成的，其中除了最左边的那个小查询以外，其余的小查询的 select_type 值就是 UNION ，可以对比上一个例子的效果，这就不多举例子了。

- UNION RESULT

  MySQL 选择使用临时表来完成 UNION 查询的去重工作，针对该临时表的查询的 select_type 就是 UNION RESULT ，例子上边有，就不赘述了。

- SUBQUERY

  如果包含子查询的查询语句不能够转为对应的 semi-join 的形式，并且该子查询是不相关子查询，并且查询优化器决定采用将该子查询物化的方案来执行该子查询时，该子查询的第一个 SELECT 关键字代表的那个查询的 select_type 就是 SUBQUERY ，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
+----+-------------+-------+------------+-------+---------------+----------+--------
-+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len
| ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+--------
-+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL
| NULL | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303
| NULL | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+--------
-+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

  可以看到，外层查询的 select_type 就是 PRIMARY ，子查询的 select_type 就是 SUBQUERY 。需要大家注意的是，由于select_type为SUBQUERY的子查询由于会被物化，所以只需要执行一遍。

- DEPENDENT SUBQUERY

  如果包含子查询的查询语句不能够转为对应的 semi-join 的形式，并且该子查询是相关子查询，则该子查询的第一个 SELECT 关键字代表的那个查询的 select_type 就是 DEPENDENT SUBQUERY ，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE s1.key2 = s
2.key2) OR key3 = 'a';
+----+--------------------+-------+------------+------+-------------------+---------
-+---------+-------------------+------+----------+-------------+
| id | select_type        | table | partitions | type | possible_keys     | key
| key_len | ref               | rows | filtered | Extra       |
+----+--------------------+-------+------------+------+-------------------+---------
-+---------+-------------------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL  | idx_key3          | NULL
| NULL    | NULL              | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | ref  | idx_key2,idx_key1 | idx_key2
| 5       | xiaohaizi.s1.key2 |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+------+-------------------+---------
-+---------+-------------------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

  需要大家注意的是，select_type为DEPENDENT SUBQUERY的查询可能会被执行多次。

- DEPENDENT UNION

在包含 UNION 或者 UNION ALL 的大查询中，如果各个小查询都依赖于外层查询的话，那除了最左边的那个小查询之外，其余的小查询的 select_type 的值就是 DEPENDENT UNION 。说的有些绕哈，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE key1 = 'a' UNION SELECT key1 FROM s1 WHERE key1 = 'b');
+----+--------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+--------------------------+
| id | select_type        | table      | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra                    |
+----+--------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+--------------------------+
|  1 | PRIMARY            | s1         | NULL       | ALL  | NULL          | NULL     | NULL    | NULL  | 9688 |   100.00 | Using where              |
|  2 | DEPENDENT SUBQUERY | s2         | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |   12 |   100.00 | Using where; Using index |
|  3 | DEPENDENT UNION    | s1         | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | Using where; Using index |
| NULL | UNION RESULT     | <union2,3> | NULL       | ALL  | NULL          | NULL     | NULL    | NULL  | NULL |     NULL | Using temporary          |
+----+--------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+--------------------------+
4 rows in set, 1 warning (0.03 sec)
```

这个查询比较复杂啊，大查询里包含了一个子查询，子查询里又是由 UNION 连起来的两个小查询。从执行计划中可以看出来， SELECT key1 FROM s2 WHERE key1 = 'a' 这个小查询由于是子查询中第一个查询，所以它的 select_type 是 DEPENDENT SUBQUERY ，而 SELECT key1 FROM s1 WHERE key1 = 'b' 这个查询的 select_type 就是 DEPENDENT UNION 。

- DERIVED

对于采用物化的方式执行的包含派生表的查询，该派生表对应的子查询的 select_type 就是 DERIVED ，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM (SELECT key1, count(*) as c FROM s1 GROUP BY key1) AS derived_s1 where c > 1;
+----+-------------+------------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table      | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+------------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL   | NULL          | NULL     | NULL    | NULL | 9688 |    33.33 | Using where |
|  2 | DERIVED     | s1         | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9688 |   100.00 | Using index |
+----+-------------+------------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

从执行计划中可以看出， id 为 2 的记录就代表子查询的执行方式，它的 select_type 是 DERIVED ，说明该子查询是以物化的方式执行的。 id 为 1 的记录代表外层查询，大家注意看它的 table 列显示的是 <derived2> ，表示该查询是针对将派生表物化之后的表进行查询的。

小贴士：
如果派生表可以通过和外层查询合并的方式执行的话，执行计划又是另一番景象，大家可以试试哈～

- MATERIALIZED

  当查询优化器在执行包含子查询的语句时，选择将子查询物化之后与外层查询进行连接查询时，该子查询对应的 select_type 属性就是 MATERIALIZED ，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2);
+----+--------------+-------------+------------+--------+---------------+------------
-+---------+-------------------+------+----------+-------------+
| id | select_type  | table       | partitions | type   | possible_keys | key
| key_len | ref               | rows | filtered | Extra       |
+----+--------------+-------------+------------+--------+---------------+------------
-+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE       | s1          | NULL       | ALL    | idx_key1      | NULL
| NULL    | NULL              | 9688 |   100.00 | Using where |
|  1 | SIMPLE       | <subquery2> | NULL       | eq_ref | <auto_key>    | <auto_key>
| 303     | xiaohaizi.s1.key1 |    1 |   100.00 | NULL        |
|  2 | MATERIALIZED | s2          | NULL       | index  | idx_key1      | idx_key1
| 303     | NULL              | 9954 |   100.00 | Using index |
+----+--------------+-------------+------------+--------+---------------+------------
-+---------+-------------------+------+----------+-------------+
3 rows in set, 1 warning (0.01 sec)
```

  执行计划的第三条记录的 id 值为 2 ，说明该条记录对应的是一个单表查询，从它的 select_type 值为 MATERIALIZED 可以看出，查询优化器是要把子查询先转换成物化表。然后看执行计划的前两条记录的 id 值都为 1 ，说明这两条记录对应的表进行连接查询，需要注意的是第二条记录的 table 列的值是 <subquery2> ，说明该表其实就是 id 为 2 对应的子查询执行之后产生的物化表，然后将 s1 和该物化表进行连接查询。

- UNCACHEABLE SUBQUERY

  不常用，就不多唠叨了。

- UNCACHEABLE UNION

  不常用，就不多唠叨了。

15.1.4 partitions

由于我们压根儿就没唠叨过分区是个啥，所以这个输出列我们也就不说了哈，一般情况下我们的查询语句的执行计划的 partitions 列的值都是 NULL 。

15.1.5 type

我们前边说过执行计划的一条记录就代表着 MySQL 对某个表的执行查询时的访问方法，其中的 type 列就表明了这个访问方法是个啥，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.04 sec)
```

可以看到 `type` 列的值是 `ref` ，表明 MySQL 即将使用 `ref` 访问方法来执行对 `s1` 表的查询。但是我们之前只唠叨过对使用 InnoDB 存储引擎的表进行单表访问的一些访问方法，完整的访问方法如下： `system` ， `const` ， `eq_ref` ， `ref` ， `fulltext` ， `ref_or_null` ， `index_merge` ， `unique_subquery` ， `index_subquery` ， `range` ， `index` ， `ALL` 。当然我们还要详细唠叨一下哈：

- `system`

  当表中只有一条记录并且该表使用的存储引擎的统计数据是精确的，比如MyISAM、Memory，那么对该表的访问方法就是 `system` 。比方说我们新建一个 MyISAM 表，并为其插入一条记录：

  ```
  mysql> CREATE TABLE t(i int) Engine=MyISAM;
  Query OK, 0 rows affected (0.05 sec)

  mysql> INSERT INTO t VALUES(1);
  Query OK, 1 row affected (0.01 sec)
  ```

  然后我们看一下查询这个表的执行计划：

  ```
  mysql> EXPLAIN SELECT * FROM t;
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  | id | select_type | table | partitions | type   | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  |  1 | SIMPLE      | t     | NULL       | system | NULL          | NULL | NULL    | NULL |    1 |   100.00 | NULL  |
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  1 row in set, 1 warning (0.00 sec)
  ```

  可以看到 `type` 列的值就是 `system` 了。

  > 小贴士：
  > 你可以把表改成使用InnoDB存储引擎，试试看执行计划的type列是什么。

- `const`

  这个我们前边唠叨过，就是当我们根据主键或者唯一二级索引列与常数进行等值匹配时，对单表的访问方法就是 `const` ，比如：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 5;
+----+-------------+-------+------------+-------+---------------+---------+---------
+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len
| ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------
+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 4
| const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+---------+---------
+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

- eq_ref

在连接查询时，如果被驱动表是通过主键或者唯一二级索引列等值匹配的方式进行访问的（如果该主键或者唯一二级索引是联合索引的话，所有的索引列都必须进行等值比较），则对该被驱动表的访问方法就是 eq_ref ，比方说：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+----+-------------+-------+------------+--------+---------------+---------+--------
-+-----------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len
| ref             | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+--------
-+-----------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL    | PRIMARY       | NULL    | NULL
| NULL            | 9688 |   100.00 | NULL  |
|  1 | SIMPLE      | s2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4
| xiaohaizi.s1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+--------
-+-----------------+------+----------+-------+
2 rows in set, 1 warning (0.01 sec)
```

从执行计划的结果中可以看出， MySQL 打算将 s1 作为驱动表， s2 作为被驱动表，重点关注 s2 的访问方法是 eq_ref ，表明在访问 s2 表的时候可以通过主键的等值匹配来进行访问。

- ref

当通过普通的二级索引列与常量进行等值匹配时来查询某个表，那么对该表的访问方法就可能是 ref ，最开始举过例子了，就不重复举例了。

- fulltext

全文索引，我们没有细讲过，跳过~

- ref_or_null

当对普通二级索引进行等值匹配查询，该索引列的值也可以是 NULL 值时，那么对该表的访问方法就可能是 ref_or_null ，比如说：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key1 IS NULL;
+----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
| id | select_type | table | partitions | type        | possible_keys | key      | key_len | ref   | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | ref_or_null | idx_key1      | idx_key1 | 303     | const |    9 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

- index_merge

  一般情况下对于某个表的查询只能使用到一个索引，但我们唠叨单表访问方法时特意强调了在某些场景下可以使用 Intersection 、 Union 、 Sort-Union 这三种索引合并的方式来执行查询，忘掉的回去补一下哈，我们看一下执行计划中是怎么体现 MySQL 使用索引合并的方式来对某个表执行查询的：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';
+----+-------------+-------+------------+-------------+-------------------+-------------------+---------+------+------+----------+---------------------------------------------+
| id | select_type | table | partitions | type        | possible_keys     | key               | key_len | ref  | rows | filtered | Extra                                       |
+----+-------------+-------+------------+-------------+-------------------+-------------------+---------+------+------+----------+---------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | index_merge | idx_key1,idx_key3 | idx_key1,idx_key3 | 303,303 | NULL |   14 |   100.00 | Using union(idx_key1,idx_key3); Using where |
+----+-------------+-------+------------+-------------+-------------------+-------------------+---------+------+------+----------+---------------------------------------------+
1 row in set, 1 warning (0.01 sec)
```

  从执行计划的 type 列的值是 index_merge 就可以看出， MySQL 打算使用索引合并的方式来执行对 s1 表的查询。

- unique_subquery

  类似于两表连接中被驱动表的 eq_ref 访问方法， unique_subquery 是针对在一些包含 IN 子查询的查询语句中，如果查询优化器决定将 IN 子查询转换为 EXISTS 子查询，而且子查询可以使用到主键进行等值匹配的话，那么该子查询执行计划的 type 列的值就是 unique_subquery ，比如下边的这个查询语句：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 IN (SELECT id FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
+----+--------------------+-------+------------+-----------------+------------------+---------+---------+------+------+----------+-------------+
| id | select_type        | table | partitions | type            | possible_keys    | key     | key_len | ref  | rows | filtered | Extra       |
+----+--------------------+-------+------------+-----------------+------------------+---------+---------+------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL             | idx_key3         | NULL    | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | unique_subquery | PRIMARY,idx_key1 | PRIMARY | 4       | func |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+-----------------+------------------+---------+---------+------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

可以看到执行计划的第二条记录的 type 值就是 unique_subquery ，说明在执行子查询时会使用到 id 列的索引。

- index_subquery

  index_subquery 与 unique_subquery 类似，只不过访问子查询中的表时使用的是普通的索引，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field IN (SELECT key3 FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
+----+--------------------+-------+------------+----------------+-------------------+----------+---------+------+------+----------+-------------+
| id | select_type        | table | partitions | type           | possible_keys     | key      | key_len | ref  | rows | filtered | Extra       |
+----+--------------------+-------+------------+----------------+-------------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL            | idx_key3          | NULL     | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | index_subquery | idx_key1,idx_key3 | idx_key3 | 303     | func |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+----------------+-------------------+----------+---------+------+------+----------+-------------+
2 rows in set, 2 warnings (0.01 sec)
```

- range

如果使用索引获取某些 范围区间 的记录，那么就可能使用到 range 访问方法，比如下边的这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN ('a', 'b', 'c');
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |   27 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

或者:

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'a' AND key1 < 'b';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  294 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

- index

当我们可以使用索引覆盖，但需要扫描全部的索引记录时，该表的访问方法就是 index ，比如这样:

```
mysql> EXPLAIN SELECT key_part2 FROM s1 WHERE key_part3 = 'a';
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key_part | 909     | NULL | 9688 |    10.00 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

上述查询中的搜索列表中只有 key_part2 一个列，而且搜索条件中也只有 key_part3 一个列，这两个列又恰好包含在 idx_key_part 这个索引中，可是搜索条件 key_part3 不能直接使用该索引进行 ref 或者 range 方式的访问，只能扫描整个 idx_key_part 索引的记录，所以查询计划的 type 列的值就是 index 。

> 小贴士：
> 再一次强调，对于使用InnoDB存储引擎的表来说，二级索引的记录只包含索引列和主键列的值，而聚簇索引中包含用户定义的全部列以及一些隐藏列，所以扫描二级索引的代价比直接全表扫描，也就是扫描聚簇索引的代价更低一些。

- ALL

最熟悉的全表扫描，就不多唠叨了，直接看例子:

```
mysql> EXPLAIN SELECT * FROM s1;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

一般来说，这些访问方法按照我们介绍它们的顺序性能依次变差。其中除了 All 这个访问方法外，其余的访问方法都能用到索引，除了 index_merge 访问方法外，其余的访问方法都最多只能用到一个索引。

15.1.6 possible_keys和key

在 EXPLAIN 语句输出的执行计划中， possible_keys 列表示在某个查询语句中，对某个表执行单表查询时可能用到的索引有哪些， key 列表示实际用到的索引有哪些，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key3 = 'a';
+----+-------------+-------+------------+------+-------------------+----------+---------+-
------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys     | key      | key_len |
 ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+-------------------+----------+---------+-
------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1,idx_key3 | idx_key3 | 303     |
 const |    6 |     2.75 | Using where |
+----+-------------+-------+------------+------+-------------------+----------+---------+-
------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

上述执行计划的 possible_keys 列的值是 idx_key1,idx_key3 ，表示该查询可能使用到 idx_key1,idx_key3 两个索引，然后 key 列的值是 idx_key3 ，表示经过查询优化器计算使用不同索引的成本后，最后决定使用 idx_key3 来执行查询比较划算。

不过有一点比较特别，就是在使用 index 访问方法来查询某个表时， possible_keys 列是空的，而 key 列展示的是实际使用到的索引，比如这样：

```
mysql> EXPLAIN SELECT key_part2 FROM s1 WHERE key_part3 = 'a';
+----+-------------+-------+------------+-------+---------------+--------------+---------+
------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len |
ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+---------------+--------------+---------+
------+------+----------+--------------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key_part | 909     |
NULL | 9688 |    10.00 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+--------------+---------+
------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

另外需要注意的一点是，possible_keys列中的值并不是越多越好，可能使用的索引越多，查询优化器计算查询成本时就得花费更长时间，所以如果可以的话，尽量删除那些用不到的索引。

15.1.7 key_len

key_len 列表示当优化器决定使用某个索引执行查询时，该索引记录的最大长度，它是由这三个部分构成的：

- 对于使用固定长度类型的索引列来说，它实际占用的存储空间的最大长度就是该固定值，对于指定字符集的变长类型的索引列来说，比如某个索引列的类型是 VARCHAR(100) ，使用的字符集是 utf8 ，那么该列实际占用的最大存储空间就是 100 × 3 = 300 个字节。
- 如果该索引列可以存储 NULL 值，则 key_len 比不可以存储 NULL 值时多1个字节。
- 对于变长字段来说，都会有2个字节的空间来存储该变长列的实际长度。

比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 5;
+----+-------------+-------+------------+-------+---------------+---------+---------+-----
--+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref
| rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-----
--+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 4       | cons
t |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+---------+---------+-----
--+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

由于 id 列的类型是 INT ，并且不可以存储 NULL 值，所以在使用该列的索引时 key_len 大小就是 4 。当索引列可以存储 NULL 值时，比如：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 = 5;
+----+-------------+-------+------------+-------+---------------+----------+---------+----
---+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref
| rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
---+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | idx_key2      | idx_key2 | 5       | con
st |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
---+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

可以看到 key_len 列就变成了 5 ，比使用 id 列的索引时多了 1 。

对于可变长度的索引列来说，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref
| rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | cons
t |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

由于 key1 列的类型是 VARCHAR(100) ，所以该列实际最多占用的存储空间就是 300 字节，又因为该列允许存储 NULL 值，所以 key_len 需要加 1 ，又因为该列是可变长度列，所以 key_len 需要加 2 ，所以最后 ken_len 的值就是 303 。

有的同学可能有疑问：你在前边唠叨 InnoDB 行格式的时候不是说，存储变长字段的实际长度不是可能占用1个字节或者2个字节么？为什么现在不管三七二十一都用了 2 个字节？这里需要强调的一点是，执行计划的生成是在 MySQL server 层中的功能，并不是针对具体某个存储引擎的功能，设计 MySQL 的大叔在执行计划中输出 key_len 列主要是为了让我们区分某个使用联合索引的查询具体用了几个索引列，而不是为了准确的说明针对某个具体存储引擎存储变长字段的实际长度占用的空间到底是占用1个字节还是2个字节。比方说下边这个使用到联合索引 idx_key_part 的查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a';
+----+-------------+-------+------------+------+---------------+--------------+---------+-
------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len |
 ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+--------------+---------+-
------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key_part  | idx_key_part | 303     |
 const |   12 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+--------------+---------+-
------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

我们可以从执行计划的 key_len 列中看到值是 303 ，这意味着 MySQL 在执行上述查询中只能用到 idx_key_part 索引的一个索引列，而下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a' AND key_part2 = 'b';
+----+-------------+-------+------------+------+---------------+--------------+---------+-
------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len |
 ref         | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+--------------+---------+-
------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key_part  | idx_key_part | 606     |
 const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+--------------+---------+-
------------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

这个查询的执行计划的 ken_len 列的值是 606 ，说明执行这个查询的时候可以用到联合索引 idx_key_part 的两个索引列。

15.1.8 ref

当使用索引列等值匹配的条件去执行查询时，也就是在访问方法是 const 、 eq_ref 、 ref 、 ref_or_null 、 unique_subquery 、 index_subquery 其中之一时， ref 列展示的就是与索引列作等值匹配的东东是个啥，比如只是一个常数或者是某个列。大家看下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref
| rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | cons
t |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

可以看到 ref 列的值是 const ，表明在使用 idx_key1 索引执行查询时，与 key1 列作等值匹配的对象是一个常数，当然有时候更复杂一点：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+----+-------------+-------+------------+--------+---------------+---------+---------+----
------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref
| rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+----
------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NUL
L            | 9688 |   100.00 | NULL  |
|  1 | SIMPLE      | s2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | xia
ohaizi.s1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+----
------------+------+----------+-------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到对被驱动表 s2 的访问方法是 eq_ref ，而对应的 ref 列的值是 xiaohaizi.s1.id ，这说明在对被驱动表进行访问时会用到 PRIMARY 索引，也就是聚簇索引与一个列进行等值匹配的条件，于 s2 表的 id 作等值匹配的对象就是 xiaohaizi.s1.id 列（注意这里把数据库名也写出来了）。

有的时候与索引列进行等值匹配的对象是一个函数，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s2.key1 = UPPER(s1.key1);
+----+-------------+-------+------------+------+---------------+----------+---------+-----
-+------+----------+-----------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref
| rows | filtered | Extra                 |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
-+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL     | NULL    | NULL
| 9688 |   100.00 | NULL                  |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303     | func
|    1 |   100.00 | Using index condition |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
-+------+----------+-----------------------+
2 rows in set, 1 warning (0.00 sec)
```

我们看执行计划的第二条记录，可以看到对 s2 表采用 ref 访问方法执行查询，然后在查询计划的 ref 列里输出的是 func ，说明与 s2 表的 key1 列进行等值匹配的对象是一个函数。

15.1.9 rows

如果查询优化器决定使用全表扫描的方式对某个表执行查询时，执行计划的 rows 列就代表预计需要扫描的行数，如果使用索引来执行查询时，执行计划的 rows 列就代表预计扫描的索引记录行数。比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z';
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref
| rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NUL
L |  266 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

我们看到执行计划的 rows 列的值是 266 ，这意味着查询优化器在经过分析使用 idx_key1 进行查询的成本之后，觉得满足 key1 > 'z' 这个条件的记录只有 266 条。

15.1.10 filtered

之前在分析连接查询的成本时提出过一个 condition filtering 的概念，就是 MySQL 在计算驱动表扇出时采用的一个策略：

- 如果使用的是全表扫描的方式执行的单表查询，那么计算驱动表扇出时需要估计出满足搜索条件的记录到底有多少条。
- 如果使用的是索引执行的单表扫描，那么计算驱动表扇出的时候需要估计出满足除使用到对应索引的搜索条件外的其他搜索条件的记录有多少条。

比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND common_field = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref
| rows | filtered | Extra                              |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NUL
L |  266 |    10.00 | Using index condition; Using where |
+----+-------------+-------+------------+-------+---------------+----------+---------+----
--+------+----------+------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

从执行计划的 key 列中可以看出来，该查询使用 idx_key1 索引来执行查询，从 rows 列可以看出满足 key1 > 'z' 的记录有 266 条。执行计划的 filtered 列就代表查询优化器预测在这 266 条记录中，有多少条记录满足其余的搜索条件，也就是 common_field = 'a' 这个条件的百分比。此处 filtered 列的值是 10.00 ，说明查询优化器预测在 266 条记录中有 10.00% 的记录满足 common_field = 'a' 这个条件。

对于单表查询来说，这个 filtered 列的值没什么意义，我们更关注在连接查询中驱动表对应的执行计划记录的 filtered 值，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key1 WHERE s1.common_field =
'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref
| rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--------------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL
| 9688 |    10.00 | Using where |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiao
haizi.s1.key1 |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

从执行计划中可以看出来，查询优化器打算把 s1 当作驱动表， s2 当作被驱动表。我们可以看到驱动表 s1 表的执行计划的 rows 列为 9688 ， filtered 列为 10.00 ，这意味着驱动表 s1 的扇出值就是 9688 × 10.00% = 968.8 ，这说明还要对被驱动表执行大约 968 次查询。

16 第16章 查询优化的百科全书-Explain详解（下）

标签： MySQL 是怎样运行的

16.1 执行计划输出中各列详解

本章紧接着上一节的内容，继续唠叨 EXPLAIN 语句输出的各个列的意思。

16.1.1 Extra

顾名思义， Extra 列是用来说明一些额外信息的，我们可以通过这些额外信息来更准确的理解 MySQL 到底将如何执行给定的查询语句。 MySQL 提供的额外信息有好几十个，我们就不一个一个介绍了（都介绍了感觉我们的文章就跟文档差不多了～），所以我们只挑一些平时常见的或者比较重要的额外信息介绍给大家哈。

- No tables used

  当查询语句的没有 FROM 子句时将会提示该额外信息，比如：

```
  mysql> EXPLAIN SELECT 1;
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra          |
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------+
  |  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NU
LL | NULL |     NULL | No tables used |
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------+
  1 row in set, 1 warning (0.00 sec)
```

- Impossible WHERE

  查询语句的 WHERE 子句永远为 FALSE 时将会提示该额外信息，比方说：

```
  mysql> EXPLAIN SELECT * FROM s1 WHERE 1 != 1;
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+------------------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra            |
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+------------------+
  |  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NU
LL | NULL |     NULL | Impossible WHERE |
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+------------------+
  1 row in set, 1 warning (0.01 sec)
```

- No matching min/max row

  当查询列表处有 MIN 或者 MAX 聚集函数，但是并没有符合 WHERE 子句中的搜索条件的记录时，将会提示该额外信息，比方说：

```
  mysql> EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'abcdefg';
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------------------------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra                   |
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------------------------+
  |  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NU
LL | NULL |     NULL | No matching min/max row |
  +----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-------------------------+
  1 row in set, 1 warning (0.00 sec)
```

1 row in set, 1 warning (0.00 sec)

- Using index

  当我们的查询列表以及搜索条件中只包含属于某个索引的列，也就是在可以使用索引覆盖的情况下，在 Extra 列将会提示该额外信息。比方说下边这个查询中只需要用到 idx_key1 而不需要回表操作：

```
mysql> EXPLAIN SELECT key1 FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------
+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len
| ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------
+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303
| const |    8 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------
+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- Using index condition

  有些搜索条件中虽然出现了索引列，但却不能使用到索引，比如下边这个查询：

```
SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%a';
```

  其中的 key1 > 'z' 可以使用到索引，但是 key1 LIKE '%a' 却无法使用到索引，在以前版本的 MySQL 中，是按照下边步骤来执行这个查询的：
  - 先根据 key1 > 'z' 这个条件，从二级索引 idx_key1 中获取到对应的二级索引记录。
  - 根据上一步骤得到的二级索引记录中的主键值进行回表，找到完整的用户记录再检测该记录是否符合 key1 LIKE '%a' 这个条件，将符合条件的记录加入到最后的结果集。

    但是虽然 key1 LIKE '%a' 不能组成范围区间参与 range 访问方法的执行，但这个条件毕竟只涉及到了 key1 列，所以设计 MySQL 的大叔把上边的步骤改进了一下：
  - 先根据 key1 > 'z' 这个条件，定位到二级索引 idx_key1 中对应的二级索引记录。
  - 对于指定的二级索引记录，先不着急回表，而是先检测一下该记录是否满足 key1 LIKE '%a' 这个条件，如果这个条件不满足，则该二级索引记录压根儿就没必要回表。
  - 对于满足 key1 LIKE '%a' 这个条件的二级索引记录执行回表操作。

    我们说回表操作其实是一个随机 IO ，比较耗时，所以上述修改虽然只改进了一点点，但是可以省去好多回表操作的成本。设计 MySQL 的大叔们把他们的这个改进称之为 索引条件下推 （英文名： Index Condition Pushdown ）。

    如果在查询语句的执行过程中将要使用 索引条件下推 这个特性，在 Extra 列中将会显示 Using index condition ，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%b';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

- Using where

当我们使用全表扫描来执行对某个表的查询，并且该语句的 WHERE 子句中有针对该表的搜索条件时，在 Extra 列中会提示上述额外信息。比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field = 'a';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

当使用索引访问来执行对某个表的查询，并且该语句的 WHERE 子句中有除了该索引包含的列之外的其他搜索条件时，在 Extra 列中也会提示上述额外信息。比如下边这个查询虽然使用 idx_key1 索引执行查询，但是搜索条件中除了包含 key1 的搜索条件 key1 = 'a' ，还有包含 common_field 的搜索条件，所以 Extra 列会显示 Using where 的提示：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND common_field = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- Using join buffer (Block Nested Loop)

在连接查询执行过程中，当被驱动表不能有效的利用索引加快访问速度， MySQL 一般会为其分配一块名叫 join buffer 的内存块来加快查询速度，也就是我们所讲的 基于块的嵌套循环算法 ，比如下边这个查询语句：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.common_field = s2.common_field;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9954 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.03 sec)
```

可以在对 s2 表的执行计划的 Extra 列显示了两个提示：

- Using join buffer (Block Nested Loop)：这是因为对表 s2 的访问不能有效利用索引，只好退而求其次，使用 join buffer 来减少对 s2 表的访问次数，从而提高性能。
- Using where：可以看到查询语句中有一个 s1.common_field = s2.common_field 条件，因为 s1 是驱动表， s2 是被驱动表，所以在访问 s2 表时， s1.common_field 的值已经确定下来了，所以实际上查询 s2 表的条件就是 s2.common_field = 一个常数，所以提示了 Using where 额外信息。

- Not exists

当我们使用左（外）连接时，如果 WHERE 子句中包含要求被驱动表的某个列等于 NULL 值的搜索条件，而且那个列又是不允许存储 NULL 值的，那么在该表的执行计划的 Extra 列就会提示 Not exists 额外信息，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE s2.id IS NUL
L;
+----+-------------+-------+------------+------+---------------+----------+---------
+-------------------+------+----------+-------------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len
| ref               | rows | filtered | Extra                   |
+----+-------------+-------+------------+------+---------------+----------+---------
+-------------------+------+----------+-------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL     | NULL
| NULL              | 9688 |   100.00 | NULL                    |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303
| xiaohaizi.s1.key1 |    1 |    10.00 | Using where; Not exists |
+----+-------------+-------+------------+------+---------------+----------+---------
+-------------------+------+----------+-------------------------+
2 rows in set, 1 warning (0.00 sec)
```

上述查询中 s1 表是驱动表， s2 表是被驱动表， s2.id 列是不允许存储 NULL 值的，而 WHERE 子句中又包含 s2.id IS NULL 的搜索条件，这意味着必定是驱动表的记录在被驱动表中找不到匹配 ON 子句条件的记录才会把该驱动表的记录加入到最终的结果集，所以对于某条驱动表中的记录来说，如果能在被驱动表中找到1条符合 ON 子句条件的记录，那么该驱动表的记录就不会被加入到最终的结果集，也就是说我们没有必要到被驱动表中找到全部符合ON子句条件的记录，这样可以稍微节省一点性能。

```
小贴士：
右（外）连接可以被转换为左（外）连接，所以就不提右（外）连接的情况了。
```

- Using intersect(...) 、 Using union(...) 和 Using sort_union(...)

  如果执行计划的 Extra 列出现了 Using intersect(...) 提示，说明准备使用 Intersect 索引合并的方式执行查询，括号中的 ... 表示需要进行索引合并的索引名称；如果出现了 Using union(...) 提示，说明准备使用 Union 索引合并的方式执行查询；出现了 Using sort_union(...) 提示，说明准备使用 Sort-Union 索引合并的方式执行查询。比如这个查询的执行计划：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND key3 = 'a';
+----+-------------+-------+------------+-------------+-------------------+---------------------+---------+------+------+----------+-------------------------------------------------+
| id | select_type | table | partitions | type        | possible_keys     | key                 | key_len | ref  | rows | filtered | Extra                                           |
+----+-------------+-------+------------+-------------+-------------------+---------------------+---------+------+------+----------+-------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | index_merge | idx_key1,idx_key3 | idx_key3,idx_key1 | 303,303 | NULL |    1 |   100.00 | Using intersect(idx_key3,idx_key1); Using where |
+----+-------------+-------+------------+-------------+-------------------+---------------------+---------+------+------+----------+-------------------------------------------------+
1 row in set, 1 warning (0.01 sec)
```

  其中 Extra 列就显示了 Using intersect(idx_key3,idx_key1) ，表明 MySQL 即将使用 idx_key3 和 idx_key1 这两个索引进行 Intersect 索引合并的方式执行查询。

  > 小贴士：
  > 剩下两种类型的索引合并的Extra列信息就不一一举例子了，自己写个查询瞅瞅呗～

- Zero limit

  当我们的 LIMIT 子句的参数为 0 时，表示压根儿不打算从表中读出任何记录，将会提示该额外信息，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 LIMIT 0;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra      |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | Zero limit |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
1 row in set, 1 warning (0.00 sec)
```

- Using filesort

  有一些情况下对结果集中的记录进行排序是可以使用到索引的，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 ORDER BY key1 LIMIT 10;
+----+-------------+-------+------------+-------+---------------+----------+--------
-+------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len
| ref  | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+--------
-+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key1 | 303
| NULL |   10 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+--------
-+------+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

这个查询语句可以利用 idx_key1 索引直接取出 key1 列的10条记录，然后再进行回表操作就好了。但是很多情况下排序操作无法使用到索引，只能在内存中（记录较少的时候）或者磁盘中（记录较多的时候）进行排序，设计 MySQL 的大叔把这种在内存中或者磁盘上进行排序的方式统称为文件排序（英文名： filesort ）。如果某个查询需要使用文件排序的方式执行查询，就会在执行计划的 Extra 列中显示 Using filesort 提示，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 ORDER BY common_field LIMIT 10;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | Using filesort |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+----------------+
1 row in set, 1 warning (0.00 sec)
```

需要注意的是，如果查询中需要使用 filesort 的方式进行排序的记录非常多，那么这个过程是很耗费性能的，我们最好想办法将使用 文件排序 的执行方式改为使用索引进行排序。

- Using temporary

在许多查询的执行过程中， MySQL 可能会借助临时表来完成一些功能，比如去重、排序之类的，比如我们在执行许多包含 DISTINCT 、 GROUP BY 、 UNION 等子句的查询过程中，如果不能有效利用索引来完成查询， MySQL 很有可能寻求通过建立内部的临时表来执行查询。如果查询中使用到了内部的临时表，在执行计划的 Extra 列将会显示 Using temporary 提示，比方说这样：

```
mysql> EXPLAIN SELECT DISTINCT common_field FROM s1;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra           |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)
```

再比如：

```
mysql> EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY common_fiel
d;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+---------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra                           |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+---------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | Using temporary; Using filesort |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+---------------------------------+
1 row in set, 1 warning (0.00 sec)
```

不知道大家注意到没有，上述执行计划的 Extra 列不仅仅包含 Using temporary 提示，还包含 Using filesort 提示，可是我们的查询语句中明明没有写 ORDER BY 子句呀？这是因为 MySQL 会在包含 GROUP BY 子句的查询中默认添加上 ORDER BY 子句，也就是说上述查询其实和下边这个查询等价：

```
EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY common_field ORDER
BY common_field;
```

如果我们并不想为包含 GROUP BY 子句的查询进行排序，需要我们显式的写上 ORDER BY NULL ，就像这样：

```
mysql> EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY common_field
ORDER BY NULL;
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | re
f  | rows | filtered | Extra           |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NU
LL | 9688 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+---
---+------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)
```

这回执行计划中就没有 Using filesort 的提示了，也就意味着执行查询时可以省去对记录进行文件排序的成本了。

另外，执行计划中出现 Using temporary 并不是一个好的征兆，因为建立与维护临时表要付出很大成本的，所以我们最好能使用索引来替代掉使用临时表，比方说下边这个包含 GROUP BY 子句的查询就不需要使用临时表：

```
mysql> EXPLAIN SELECT key1, COUNT(*) AS amount FROM s1 GROUP BY key1;
+----+-------------+-------+------------+-------+---------------+----------+---------
-+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len
| ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------
-+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | index | idx_key1      | idx_key1 | 303
| NULL | 9688 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------
-+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

从 Extra 的 Using index 的提示里我们可以看出，上述查询只需要扫描 idx_key1 索引就可以搞定了，不再需要临时表了。

- Start temporary, End temporary

我们前边唠叨子查询的时候说过，查询优化器会优先尝试将 IN 子查询转换成 semi-join ，而 semi-join 又有好多种执行策略，当执行策略为 DuplicateWeedout 时，也就是通过建立临时表来实现为外层查询中的记录进行去重操作时，驱动表查询执行计划的 Extra 列将显示 Start temporary 提示，被驱动表查询执行计划的 Extra 列将显示 End temporary 提示，就是这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key3 FROM s2 WHERE common_fiel
d = 'a');
+----+-------------+-------+------------+------+---------------+----------+---------
+-------------------+------+----------+------------------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len
 | ref               | rows | filtered | Extra                        |
+----+-------------+-------+------------+------+---------------+----------+---------
+-------------------+------+----------+------------------------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key3      | NULL     | NULL
 | NULL              | 9954 |    10.00 | Using where; Start temporary |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303
 | xiaohaizi.s2.key3 |    1 |   100.00 | End temporary                |
+----+-------------+-------+------------+------+---------------+----------+---------
+-------------------+------+----------+------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

- LooseScan

在将 In 子查询转为 semi-join 时，如果采用的是 LooseScan 执行策略，则在驱动表执行计划的 Extra 列就是显示 LooseScan 提示，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key3 IN (SELECT key1 FROM s2 WHERE key1 >
'z');
+----+-------------+-------+------------+-------+---------------+----------+---------
-+-------------------+------+----------+-------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len
| ref               | rows | filtered | Extra                               |
+----+-------------+-------+------------+-------+---------------+----------+---------
-+-------------------+------+----------+-------------------------------------+
|  1 | SIMPLE      | s2    | NULL       | range | idx_key1      | idx_key1 | 303
| NULL              |  270 |   100.00 | Using where; Using index; LooseScan |
|  1 | SIMPLE      | s1    | NULL       | ref   | idx_key3      | idx_key3 | 303
| xiaohaizi.s2.key1 |    1 |   100.00 | NULL                                |
+----+-------------+-------+------------+-------+---------------+----------+---------
-+-------------------+------+----------+-------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

- FirstMatch(tbl_name)

在将 In 子查询转为 semi-join 时，如果采用的是 FirstMatch 执行策略，则在被驱动表执行计划的 Extra 列就是显示 FirstMatch(tbl_name) 提示，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field IN (SELECT key1 FROM s2 where s1.
key3 = s2.key3);
+----+-------------+-------+------------+------+-------------------+----------+-----
----+-------------------+------+----------+-----------------------------+
| id | select_type | table | partitions | type | possible_keys     | key      | key_
len | ref               | rows | filtered | Extra                       |
+----+-------------+-------+------------+------+-------------------+----------+-----
----+-------------------+------+----------+-----------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key3          | NULL     | NULL
| NULL              | 9688 |   100.00 | Using where                 |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1,idx_key3 | idx_key3 | 303
| xiaohaizi.s1.key3 |    1 |     4.87 | Using where; FirstMatch(s1) |
+----+-------------+-------+------------+------+-------------------+----------+-----
----+-------------------+------+----------+-----------------------------+
2 rows in set, 2 warnings (0.00 sec)
```

16.2 Json格式的执行计划

我们上边介绍的 EXPLAIN 语句输出中缺少了一个衡量执行计划好坏的重要属性 —— 成本。不过设计 MySQL 的大叔贴心的为我们提供了一种查看某个执行计划花费的成本的方式：

- 在 EXPLAIN 单词和真正的查询语句中间加上 FORMAT=JSON 。

这样我们就可以得到一个 json 格式的执行计划，里边儿包含该计划花费的成本，比如这样：

```
mysql> EXPLAIN FORMAT=JSON SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key2 WHERE s1.co
mmon_field = 'a'\G
*************************** 1. row ***************************

EXPLAIN: {
  "query_block": {
    "select_id": 1,      # 整个查询语句只有1个SELECT关键字，该关键字对应的id号为1
    "cost_info": {
      "query_cost": "3197.16"    # 整个查询的执行成本预计为3197.16
    },
    "nested_loop": [     # 几个表之间采用嵌套循环连接算法执行

    # 以下是参与嵌套循环连接算法的各个表的信息
      {
        "table": {
          "table_name": "s1",    # s1表是驱动表
          "access_type": "ALL",      # 访问方法为ALL，意味着使用全表扫描访问
          "possible_keys": [     # 可能使用的索引
            "idx_key1"
          ],
          "rows_examined_per_scan": 9688,    # 查询一次s1表大致需要扫描9688条记录
          "rows_produced_per_join": 968,     # 驱动表s1的扇出是968
          "filtered": "10.00",   # condition filtering代表的百分比
          "cost_info": {
            "read_cost": "1840.84",      # 稍后解释
            "eval_cost": "193.76",       # 稍后解释
            "prefix_cost": "2034.60",    # 单次查询s1表总共的成本
            "data_read_per_join": "1M"   # 读取的数据量
          },
          "used_columns": [      # 执行查询中涉及到的列
            "id",
            "key1",
            "key2",
            "key3",
            "key_part1",
            "key_part2",
            "key_part3",
            "common_field"
          ],

          # 对s1表访问时针对单表查询的条件
          "attached_condition": "((`xiaohaizi`.`s1`.`common_field` = 'a') and (`xiaohaizi
`.`s1`.`key1` is not null))"
        }
      },
      {
        "table": {
          "table_name": "s2",    # s2表是被驱动表
          "access_type": "ref",      # 访问方法为ref，意味着使用索引等值匹配的方式访问
          "possible_keys": [     # 可能使用的索引
            "idx_key2"
          ],
```

```
          "key": "idx_key2",    # 实际使用的索引
          "used_key_parts": [   # 使用到的索引列
            "key2"
          ],
          "key_length": "5",    # key_len
          "ref": [      # 与key2列进行等值匹配的对象
            "xiaohaizi.s1.key1"
          ],
          "rows_examined_per_scan": 1,  # 查询一次s2表大致需要扫描1条记录
          "rows_produced_per_join": 968,    # 被驱动表s2的扇出是968（由于后边没有多余的表进行连接，所以这个值也没啥用）
          "filtered": "100.00",     # condition filtering代表的百分比

          # s2表使用索引进行查询的搜索条件
          "index_condition": "(`xiaohaizi`.`s1`.`key1` = `xiaohaizi`.`s2`.`key2`)",
          "cost_info": {
            "read_cost": "968.80",      # 稍后解释
            "eval_cost": "193.76",      # 稍后解释
            "prefix_cost": "3197.16",   # 单次查询s1、多次查询s2表总共的成本
            "data_read_per_join": "1M"  # 读取的数据量
          },
          "used_columns": [     # 执行查询中涉及到的列
            "id",
            "key1",
            "key2",
            "key3",
            "key_part1",
            "key_part2",
            "key_part3",
            "common_field"
          ]
        }
      }
    ]
  }
}
1 row in set, 2 warnings (0.00 sec)
```

我们使用 # 后边跟随注释的形式为大家解释了 EXPLAIN FORMAT=JSON 语句的输出内容，但是大家可能有疑问 "cost_info" 里边的成本看着怪怪的，它们是怎么计算出来的？先看 s1 表的 "cost_info" 部分：

```
"cost_info": {
    "read_cost": "1840.84",
    "eval_cost": "193.76",
    "prefix_cost": "2034.60",
    "data_read_per_join": "1M"
}
```

- read_cost 是由下边这两部分组成的：
  - IO 成本
  - 检测 rows × (1 - filter) 条记录的 CPU 成本

> 小贴士：
> rows和filter都是我们前边介绍执行计划的输出列，在JSON格式的执行计划中，rows相当于rows_examined_per_scan，filtered名称不变。

- eval_cost 是这样计算的：

  检测 rows × filter 条记录的成本。

- prefix_cost 就是单独查询 s1 表的成本，也就是：

  read_cost + eval_cost

- data_read_per_join 表示在此次查询中需要读取的数据量，我们就不多唠叨这个了。

> 小贴士：
> 大家其实没必要关注MySQL为啥使用这么古怪的方式计算出read_cost和eval_cost，关注prefix_cost是查询s1表的成本就好了。

对于 s2 表的 "cost_info" 部分是这样的：

```
"cost_info": {
    "read_cost": "968.80",
    "eval_cost": "193.76",
    "prefix_cost": "3197.16",
    "data_read_per_join": "1M"
}
```

由于 s2 表是被驱动表，所以可能被读取多次，这里的 read_cost 和 eval_cost 是访问多次 s2 表后累加起来的值，大家主要关注里边儿的 prefix_cost 的值代表的是整个连接查询预计的成本，也就是单次查询 s1 表和多次查询 s2 表后的成本的和，也就是：

```
968.80 + 193.76 + 2034.60 = 3197.16
```

16.3 Extented EXPLAIN

最后，设计 MySQL 的大叔还为我们留了个彩蛋，在我们使用 EXPLAIN 语句查看了某个查询的执行计划后，紧接着还可以使用 SHOW WARNINGS 语句查看与这个查询的执行计划有关的一些扩展信息，比如这样：

```
mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE s2.
common_field IS NOT NULL;
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref
| rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--------------+------+----------+-------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL
| 9954 |    90.00 | Using where |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiao
haizi.s2.key1 |    1 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-----
--------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `xiaohaizi`.`s1`.`key1` AS `key1`,`xiaohaizi`.`s2`.`key1` A
S `key1` from `xiaohaizi`.`s1` join `xiaohaizi`.`s2` where ((`xiaohaizi`.`s1`.`key1` = `xi
aohaizi`.`s2`.`key1`) and (`xiaohaizi`.`s2`.`common_field` is not null))
1 row in set (0.00 sec)
```

大家可以看到 SHOW WARNINGS 展示出来的信息有三个字段，分别是 Level 、 Code 、 Message 。我们最常见的就是 Code 为 1003 的信息，当 Code 值为 1003 时， Message 字段展示的信息类似于查询优化器将我们的查询语句重写后的语句。比如我们上边的查询本来是一个左（外）连接查询，但是有一个 s2.common_field IS NOT NULL 的条件，着就会导致查询优化器把左（外）连接查询优化为内连接查询，从 SHOW WARNINGS 的 Message 字段也可以看出来，原本的 LEFT JOIN 已经变成了 JOIN 。

但是大家一定要注意，我们说 Message 字段展示的信息类似于查询优化器将我们的查询语句重写后的语句，并不是等价于，也就是说 Message 字段展示的信息并不是标准的查询语句，在很多情况下并不能直接拿到黑框框中运行，它只能作为帮助我们理解查 MySQL 将如何执行查询语句的一个参考依据而已。

17 第17章 神兵利器-optimizer trace表的神器功效

标签： MySQL 是怎样运行的

对于 MySQL 5.6 以及之前的版本来说，查询优化器就像是一个黑盒子一样，你只能通过 EXPLAIN 语句查看到最后优化器决定使用的执行计划，却无法知道它为什么做这个决策。这对于一部分喜欢刨根问底的小伙伴来说简直是灾难："我就觉得使用其他的执行方案比 EXPLAIN 输出的这种方案强，凭什么优化器做的决定和我想的不一样呢？"

在 MySQL 5.6 以及之后的版本中，设计 MySQL 的大叔贴心的为这部分小伙伴提出了一个 optimizer trace 的功能，这个功能可以让我们方便的查看优化器生成执行计划的整个过程，这个功能的开启与关闭由系统变量 optimizer_trace 决定，我们看一下：

```
mysql> SHOW VARIABLES LIKE 'optimizer_trace';
+-----------------+--------------------------+
| Variable_name   | Value                    |
+-----------------+--------------------------+
| optimizer_trace | enabled=off,one_line=off |
+-----------------+--------------------------+
1 row in set (0.02 sec)
```

可以看到 enabled 值为 off ，表明这个功能默认是关闭的。

> 小贴士：
> one_line的值是控制输出格式的，如果为on那么所有输出都将在一行中展示，不适合人阅读，所以我们就保持其默认值为off吧。

如果想打开这个功能，必须首先把 enabled 的值改为 on ，就像这样：

```
mysql> SET optimizer_trace="enabled=on";
Query OK, 0 rows affected (0.00 sec)
```

然后我们就可以输入我们想要查看优化过程的查询语句，当该查询语句执行完成后，就可以到 information_schema 数据库下的 OPTIMIZER_TRACE 表中查看完整的优化过程。这个 OPTIMIZER_TRACE 表有4个列，分别是：

- QUERY ：表示我们的查询语句。
- TRACE ：表示优化过程的JSON格式文本。
- MISSING_BYTES_BEYOND_MAX_MEM_SIZE ：由于优化过程可能会输出很多，如果超过某个限制时，多余的文本将不会被显示，这个字段展示了被忽略的文本字节数。
- INSUFFICIENT_PRIVILEGES ：表示是否没有权限查看优化过程，默认值是0，只有某些特殊情况下才会是 1 ，我们暂时不关心这个字段的值。

完整的使用 optimizer trace 功能的步骤总结如下：

```
# 1. 打开optimizer trace功能 (默认情况下它是关闭的):
SET optimizer_trace="enabled=on";

# 2. 这里输入你自己的查询语句
SELECT ...;

# 3. 从OPTIMIZER_TRACE表中查看上一个查询的优化过程
SELECT * FROM information_schema.OPTIMIZER_TRACE;

# 4. 可能你还要观察其他语句执行的优化过程，重复上边的第2、3步
...

# 5. 当你停止查看语句的优化过程时，把optimizer trace功能关闭
SET optimizer_trace="enabled=off";
```

现在我们有一个搜索条件比较多的查询语句，它的执行计划如下：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE
    ->     key1 > 'z' AND
    ->     key2 < 1000000 AND
    ->     key3 IN ('a', 'b', 'c') AND
    ->     common_field = 'abc';
+----+-------------+-------+------------+-------+----------------------------+----------+-
--------+------+------+----------+------------------------------------+
| id | select_type | table | partitions | type  | possible_keys              | key      |
 key_len | ref  | rows | filtered | Extra                              |
+----+-------------+-------+------------+-------+----------------------------+----------+-
--------+------+------+----------+------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key2,idx_key1,idx_key3 | idx_key2 |
 5       | NULL |   12 |     0.42 | Using index condition; Using where |
+----+-------------+-------+------------+-------+----------------------------+----------+-
--------+------+------+----------+------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

可以看到该查询可能使用到的索引有3个，那么为什么优化器最终选择了 `idx_key2` 而不选择其他的索引或者直接全表扫描呢？这时候就可以通过 `otpimzer trace` 功能来查看优化器的具体工作过程：

```
SET optimizer_trace="enabled=on";

SELECT * FROM s1 WHERE
    key1 > 'z' AND
    key2 < 1000000 AND
    key3 IN ('a', 'b', 'c') AND
    common_field = 'abc';

SELECT * FROM information_schema.OPTIMIZER_TRACE\G
```

我们直接看一下通过查询 `OPTIMIZER_TRACE` 表得到的输出（我使用 `#` 后跟随注释的形式为大家解释了优化过程中的一些比较重要的点，大家重点关注一下）：

```
*************************** 1. row ***************************
# 分析的查询语句是什么
QUERY: SELECT * FROM s1 WHERE
    key1 > 'z' AND
    key2 < 1000000 AND
    key3 IN ('a', 'b', 'c') AND
    common_field = 'abc'

# 优化的具体过程
TRACE: {
  "steps": [
    {
      "join_preparation": {     # prepare阶段
        "select#": 1,
        "steps": [
          {
            "IN_uses_bisection": true
          },
          {
            "expanded_query": "/* select#1 */ select `s1`.`id` AS `id`,`s1`.`key1` AS `key1`,`s1`.`key2` AS `key2`,`s1`.`key3` AS `key3`,`s1`.`key_part1` AS `key_part1`,`s1`.`key_part2` AS `key_part2`,`s1`.`key_part3` AS `key_part3`,`s1`.`common_field` AS `common_field` from `s1` where ((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
          }
        ] /* steps */
      } /* join_preparation */
    },
    {
      "join_optimization": {    # optimize阶段
        "select#": 1,
        "steps": [
          {
            "condition_processing": {   # 处理搜索条件
              "condition": "WHERE",
              # 原始搜索条件
              "original_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))",
              "steps": [
                {
                  # 等值传递转换
                  "transformation": "equality_propagation",
                  "resulting_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
                },
                {
                  # 常量传递转换
                  "transformation": "constant_propagation",
                  "resulting_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
                },
                {
```

```
              # 去除没用的条件
              "transformation": "trivial_condition_removal",
              "resulting_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000)
and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
            }
          ] /* steps */
        } /* condition_processing */
      },
      {
        # 替换虚拟生成列
        "substitute_generated_columns": {
        } /* substitute_generated_columns */
      },
      {
        # 表的依赖信息
        "table_dependencies": [
          {
            "table": "`s1`",
            "row_may_be_null": false,
            "map_bit": 0,
            "depends_on_map_bits": [
            ] /* depends_on_map_bits */
          }
        ] /* table_dependencies */
      },
      {
        "ref_optimizer_key_uses": [
        ] /* ref_optimizer_key_uses */
      },
      {

        # 预估不同单表访问方法的访问成本
        "rows_estimation": [
          {
            "table": "`s1`",
            "range_analysis": {
              "table_scan": {   # 全表扫描的行数以及成本
                "rows": 9688,
                "cost": 2036.7
              } /* table_scan */,

              # 分析可能使用的索引
              "potential_range_indexes": [
                {
                  "index": "PRIMARY",    # 主键不可用
                  "usable": false,
                  "cause": "not_applicable"
                },
                {
                  "index": "idx_key2",  # idx_key2可能被使用
                  "usable": true,
                  "key_parts": [
```

```
        "key2"
      ] /* key_parts */
    },
    {
      "index": "idx_key1",  # idx_key1可能被使用
      "usable": true,
      "key_parts": [
        "key1",
        "id"
      ] /* key_parts */
    },
    {
      "index": "idx_key3",  # idx_key3可能被使用
      "usable": true,
      "key_parts": [
        "key3",
        "id"
      ] /* key_parts */
    },
    {
      "index": "idx_key_part",  # idx_keypart不可用
      "usable": false,
      "cause": "not_applicable"
    }
  ] /* potential_range_indexes */,
  "setup_range_conditions": [
  ] /* setup_range_conditions */,
  "group_index_range": {
    "chosen": false,
    "cause": "not_group_by_or_distinct"
  } /* group_index_range */,

  # 分析各种可能使用的索引的成本
  "analyzing_range_alternatives": {
    "range_scan_alternatives": [
      {
        # 使用idx_key2的成本分析
        "index": "idx_key2",
        # 使用idx_key2的范围区间
        "ranges": [
          "NULL < key2 < 1000000"
        ] /* ranges */,
        "index_dives_for_eq_ranges": true,   # 是否使用index dive
        "rowid_ordered": false,     # 使用该索引获取的记录是否按照主键排序
        "using_mrr": false,     # 是否使用mrr
        "index_only": false,    # 是否是索引覆盖访问
        "rows": 12,     # 使用该索引获取的记录条数
        "cost": 15.41,  # 使用该索引的成本
        "chosen": true  # 是否选择该索引
      },
      {
        # 使用idx_key1的成本分析
```

```
        "index": "idx_key1",
        # 使用idx_key1的范围区间
        "ranges": [
          "z < key1"
        ] /* ranges */,
        "index_dives_for_eq_ranges": true,   # 同上
        "rowid_ordered": false,   # 同上
        "using_mrr": false,   # 同上
        "index_only": false,   # 同上
        "rows": 266,   # 同上
        "cost": 320.21,   # 同上
        "chosen": false,   # 同上
        "cause": "cost"   # 因为成本太大所以不选择该索引
      },
      {
        # 使用idx_key3的成本分析
        "index": "idx_key3",
        # 使用idx_key3的范围区间
        "ranges": [
          "a <= key3 <= a",
          "b <= key3 <= b",
          "c <= key3 <= c"
        ] /* ranges */,
        "index_dives_for_eq_ranges": true,   # 同上
        "rowid_ordered": false,   # 同上
        "using_mrr": false,   # 同上
        "index_only": false,   # 同上
        "rows": 21,   # 同上
        "cost": 28.21,   # 同上
        "chosen": false,   # 同上
        "cause": "cost"   # 同上
      }
    ] /* range_scan_alternatives */,

    # 分析使用索引合并的成本
    "analyzing_roworder_intersect": {
      "usable": false,
      "cause": "too_few_roworder_scans"
    } /* analyzing_roworder_intersect */
  } /* analyzing_range_alternatives */,

  # 对于上述单表查询s1最优的访问方法
  "chosen_range_access_summary": {
    "range_access_plan": {
      "type": "range_scan",
      "index": "idx_key2",
      "rows": 12,
      "ranges": [
        "NULL < key2 < 1000000"
      ] /* ranges */
    } /* range_access_plan */,
    "rows_for_plan": 12,
```

```
                "cost_for_plan": 15.41,
                "chosen": true
              } /* chosen_range_access_summary */
            } /* range_analysis */
          }
        ] /* rows_estimation */
      },
      {

        # 分析各种可能的执行计划
        #（对多表查询这可能有很多种不同的方案，单表查询的方案上边已经分析过了，直接选
取idx_key2就好）
        "considered_execution_plans": [
          {
            "plan_prefix": [
            ] /* plan_prefix */,
            "table": "`s1`",
            "best_access_path": {
              "considered_access_paths": [
                {
                  "rows_to_scan": 12,
                  "access_type": "range",
                  "range_details": {
                    "used_index": "idx_key2"
                  } /* range_details */,
                  "resulting_rows": 12,
                  "cost": 17.81,
                  "chosen": true
                }
              ] /* considered_access_paths */
            } /* best_access_path */,
            "condition_filtering_pct": 100,
            "rows_for_plan": 12,
            "cost_for_plan": 17.81,
            "chosen": true
          }
        ] /* considered_execution_plans */
      },
      {
        # 尝试给查询添加一些其他的查询条件
        "attaching_conditions_to_tables": {
          "original_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and
 (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))",
          "attached_conditions_computation": [
          ] /* attached_conditions_computation */,
          "attached_conditions_summary": [
            {
              "table": "`s1`",
              "attached": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.
`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
            }
          ] /* attached_conditions_summary */
```

```
            } /* attaching_conditions_to_tables */
          },
          {
            # 再稍稍的改进一下执行计划
            "refine_plan": [
              {
                "table": "`s1`",
                "pushed_index_condition": "(`s1`.`key2` < 1000000)",
                "table_condition_attached": "((`s1`.`key1` > 'z') and (`s1`.`key3` in
('a','b','c')) and (`s1`.`common_field` = 'abc'))"
              }
            ] /* refine_plan */
          }
        ] /* steps */
      } /* join_optimization */
    },
    {
      "join_execution": {    # execute阶段
        "select#": 1,
        "steps": [
        ] /* steps */
      } /* join_execution */
    }
  ] /* steps */
}

# 因优化过程文本太多而丢弃的文本字节大小，值为0时表示并没有丢弃
MISSING_BYTES_BEYOND_MAX_MEM_SIZE: 0

# 权限字段
INSUFFICIENT_PRIVILEGES: 0

1 row in set (0.00 sec)
```

大家看到这个输出的第一感觉就是这文本也太多了点儿吧，其实这只是优化器执行过程中的一小部分，设计 MySQL 的大叔可能会在之后的版本中添加更多的优化过程信息。不过杂乱之中其实还是蛮有规律的，优化过程大致分为了三个阶段：

- prepare 阶段
- optimize 阶段
- execute 阶段

我们所说的基于成本的优化主要集中在 optimize 阶段，对于单表查询来说，我们主要关注 optimize 阶段的 "rows_estimation" 这个过程，这个过程深入分析了对单表查询的各种执行方案的成本；对于多表连接查询来说，我们更多需要关注 "considered_execution_plans" 这个过程，这个过程里会写明各种不同的连接方式所对应的成本。反正优化器最终会选择成本最低的那种方案来作为最终的执行计划，也就是我们使用 EXPLAIN 语句所展现出的那种方案。

如果有小伙伴对使用 EXPLAIN 语句展示出的对某个查询的执行计划很不理解，大家可以尝试使用 optimizer trace 功能来详细了解每一种执行方案对应的成本，相信这个功能能让大家更深入的了解 MySQL 查询优化器。

18 第18章 调节磁盘和CPU的矛盾-InnoDB的Buffer

Pool

标签： MySQL 是怎样运行的

18.1 缓存的重要性

通过前边的唠叨我们知道，对于使用 InnoDB 作为存储引擎的表来说，不管是用于存储用户数据的索引（包括聚簇索引和二级索引），还是各种系统数据，都是以 页 的形式存放在 表空间 中的，而所谓的 表空间 只不过是 InnoDB 对文件系统上一个或几个实际文件的抽象，也就是说我们的数据说到底还是存储在磁盘上的。但是各位也都知道，磁盘的速度慢的跟乌龟一样，怎么能配得上“快如风，疾如电”的 CPU 呢？所以 InnoDB 存储引擎在处理客户端的请求时，当需要访问某个页的数据时，就会把完整的页的数据全部加载到内存中，也就是说即使我们只需要访问一个页的一条记录，那也需要先把整个页的数据加载到内存中。将整个页加载到内存中后就可以进行读写访问了，在进行完读写访问之后并不着急把该页对应的内存空间释放掉，而是将其 缓存 起来，这样将来有请求再次访问该页面时，就可以省去磁盘 IO 的开销了。

18.2 InnoDB的Buffer Pool

18.2.1 啥是个Buffer Pool

设计 InnoDB 的大叔为了缓存磁盘中的页，在 MySQL 服务器启动的时候就向操作系统申请了一片连续的内存，他们给这片内存起了个名，叫做 Buffer Pool （中文名是 缓冲池 ）。那它有多大呢？这个其实看我们机器的配置，如果你是土豪，你有 512G 内存，你分配个几百G作为 Buffer Pool 也可以啊，当然你要是没那么有钱，设置小点也行呀～ 默认情况下 Buffer Pool 只有 128M 大小。当然如果你嫌弃这个 128M 太大或者太小，可以在启动服务器的时候配置 innodb_buffer_pool_size 参数的值，它表示 Buffer Pool 的大小，就像这样：

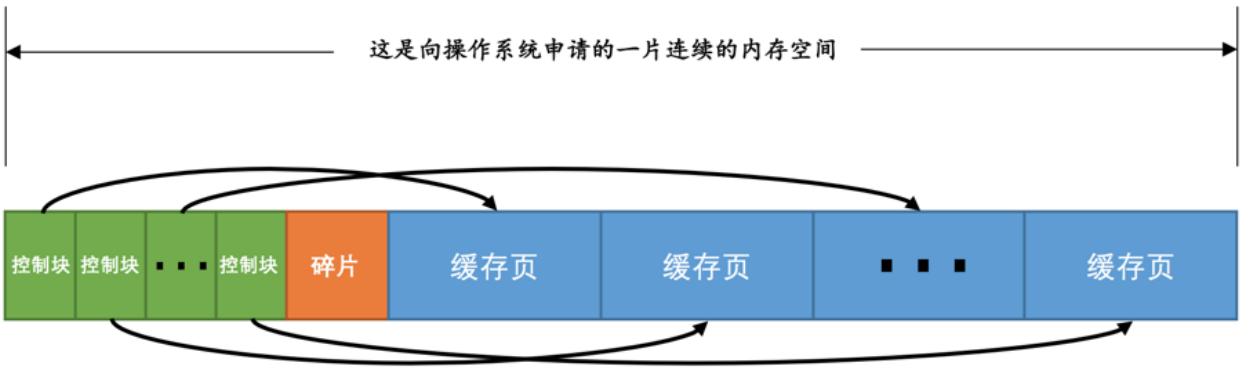
```
[server]
innodb_buffer_pool_size = 268435456
```

其中， 268435456 的单位是字节，也就是我指定 Buffer Pool 的大小为 256M 。需要注意的是， Buffer Pool 也不能太小，最小值为 5M (当小于该值时会自动设置成 5M)。

18.2.2 Buffer Pool内部组成

Buffer Pool 中默认的缓存页大小和在磁盘上默认的页大小是一样的，都是 16KB 。为了更好的管理这些在 Buffer Pool 中的缓存页，设计 InnoDB 的大叔为每一个缓存页都创建了一些所谓的 控制信息 ，这些控制信息包括该页所属的表空间编号、页号、缓存页在 Buffer Pool 中的地址、链表节点信息、一些锁信息以及 LSN 信息（锁和 LSN 我们之后会具体唠叨，现在可以先忽略），当然还有一些别的控制信息，我们这就不全唠叨一遍了，挑重要的说嘛～

每个缓存页对应的控制信息占用的内存大小是相同的，我们就把每个页对应的控制信息占用的一块内存称为一个 控制块 吧，控制块和缓存页是一一对应的，它们都被存放到 Buffer Pool 中，其中控制块被存放到 Buffer Pool 的前边，缓存页被存放到 Buffer Pool 后边，所以整个 Buffer Pool 对应的内存空间看起来就是这样的：

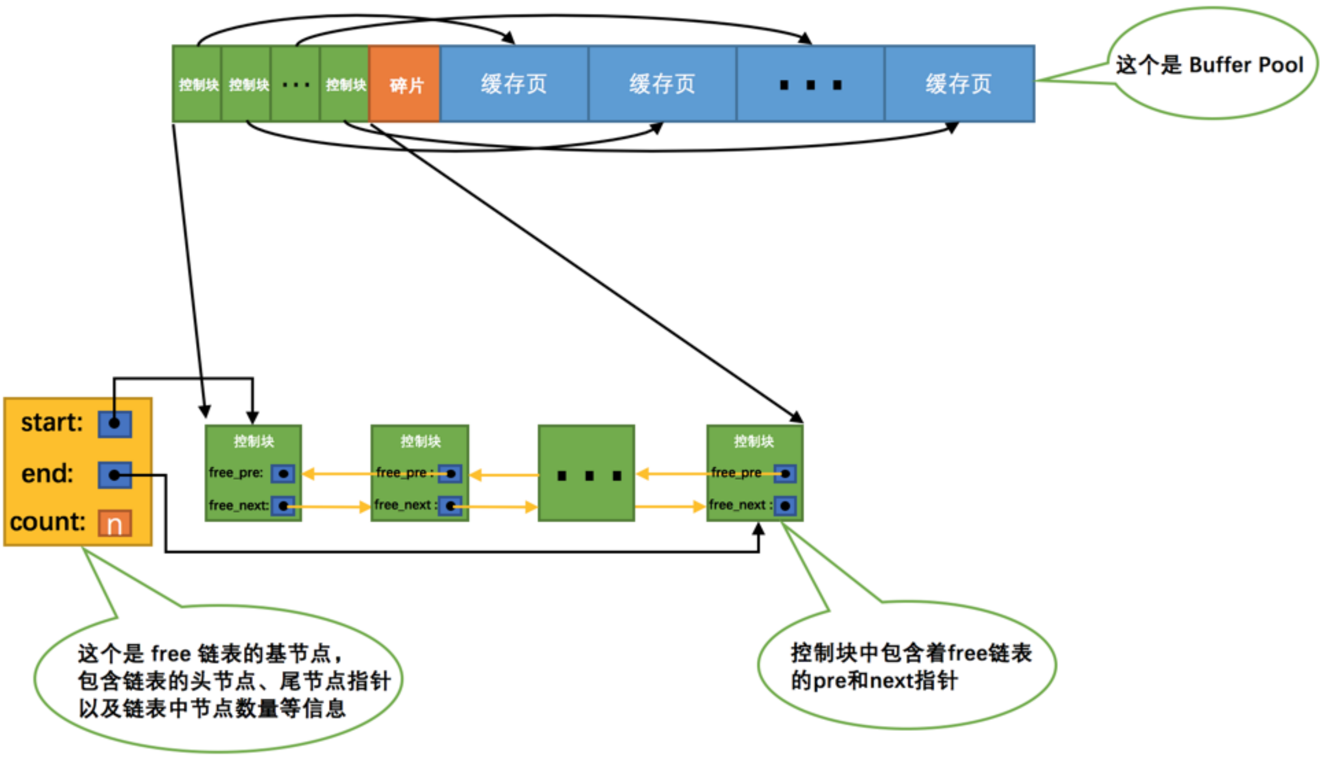


咦？控制块和缓存页之间的那个 碎片 是个什么玩意儿？你想想啊，每一个控制块都对应一个缓存页，那在分配足够多的控制块和缓存页后，可能剩余的那点儿空间不够一对控制块和缓存页的大小，自然就用不到喽，这个用不到的那点儿内存空间就被称为 碎片 了。当然，如果你把 Buffer Pool 的大小设置的刚刚好的话，也可能不会产生 碎片 ~

> 小贴士：
> 每个控制块大约占用缓存页大小的5%，在MySQL5.7.21这个版本中，每个控制块占用的大小是808字节。而我们设置的innodb_buffer_pool_size并不包含这部分控制块占用的内存空间大小，也就是说InnoDB在为Buffer Pool向操作系统申请连续的内存空间时，这片连续的内存空间一般会比innodb_buffer_pool_size的值大5%左右。

18.2.3 free链表的管理

当我们最初启动 MySQL 服务器的时候，需要完成对 Buffer Pool 的初始化过程，就是先向操作系统申请 Buffer Pool 的内存空间，然后把它划分成若干对控制块和缓存页。但是此时并没有真实的磁盘页被缓存到 Buffer Pool 中（因为还没有用到），之后随着程序的运行，会不断的有磁盘上的页被缓存到 Buffer Pool 中。那么问题来了，从磁盘上读取一个页到 Buffer Pool 中的时候该放到哪个缓存页的位置呢？或者说怎么区分 Buffer Pool 中哪些缓存页是空闲的，哪些已经被使用了呢？我们最好在某个地方记录一下Buffer Pool中哪些缓存页是可用的，这个时候缓存页对应的 控制块 就派上大用场了，我们可以把所有空闲的缓存页对应的控制块作为一个节点放到一个链表中，这个链表也可以被称作 free链表 （或者说空闲链表）。刚刚完成初始化的 Buffer Pool 中所有的缓存页都是空闲的，所以每一个缓存页对应的控制块都会被加入到 free链表 中，假设该 Buffer Pool 中可容纳的缓存页数量为 n ，那增加了 free链表 的效果图就是这样的：



从图中可以看出，我们为了管理好这个 free链表 ，特意为这个链表定义了一个 基节点 ，里边儿包含着链表的头节点地址，尾节点地址，以及当前链表中节点的数量等信息。这里需要注意的是，链表的基节点占用的内存空间并不包含在为 Buffer Pool 申请的一大片连续内存空间之内，而是单独申请的一块内存空间。

> 小贴士：
> 链表基节点占用的内存空间并不大，在MySQL5.7.21这个版本里，每个基节点只占用40字节大小。后边我们即将介绍许多不同的链表，它们的基节点和free链表的基节点的内存分配方式是一样一样的，都是单独申请的一块40字节大小的内存空间，并不包含在为Buffer Pool申请的一大片连续内存空间之内。

有了这个 free链表 之后事儿就好办了，每当需要从磁盘中加载一个页到 Buffer Pool 中时，就从 free链表 中取一个空闲的缓存页，并且把该缓存页对应的 控制块 的信息填上（就是该页所在的表空间、页号之类的信息），然后把该缓存页对应的 free链表 节点从链表中移除，表示该缓存页已经被使用了~

18.2.4 缓存页的哈希处理

我们前边说过，当我们需要访问某个页中的数据时，就会把该页从磁盘加载到 Buffer Pool 中，如果该页已经在 Buffer Pool 中的话直接使用就可以了。那么问题也就来了，我们怎么知道该页在不在 Buffer Pool 中呢？难不成需要依次遍历 Buffer Pool 中各个缓存页么？一个 Buffer Pool 中的缓存页这么多都遍历完岂不是要累死？

再回头想想，我们其实是根据 表空间号 + 页号 来定位一个页的，也就相当于 表空间号 + 页号 是一个 key ， 缓存页 就是对应的 value ，怎么通过一个 key 来快速找着一个 value 呢？哈哈，那肯定是哈希表喽~

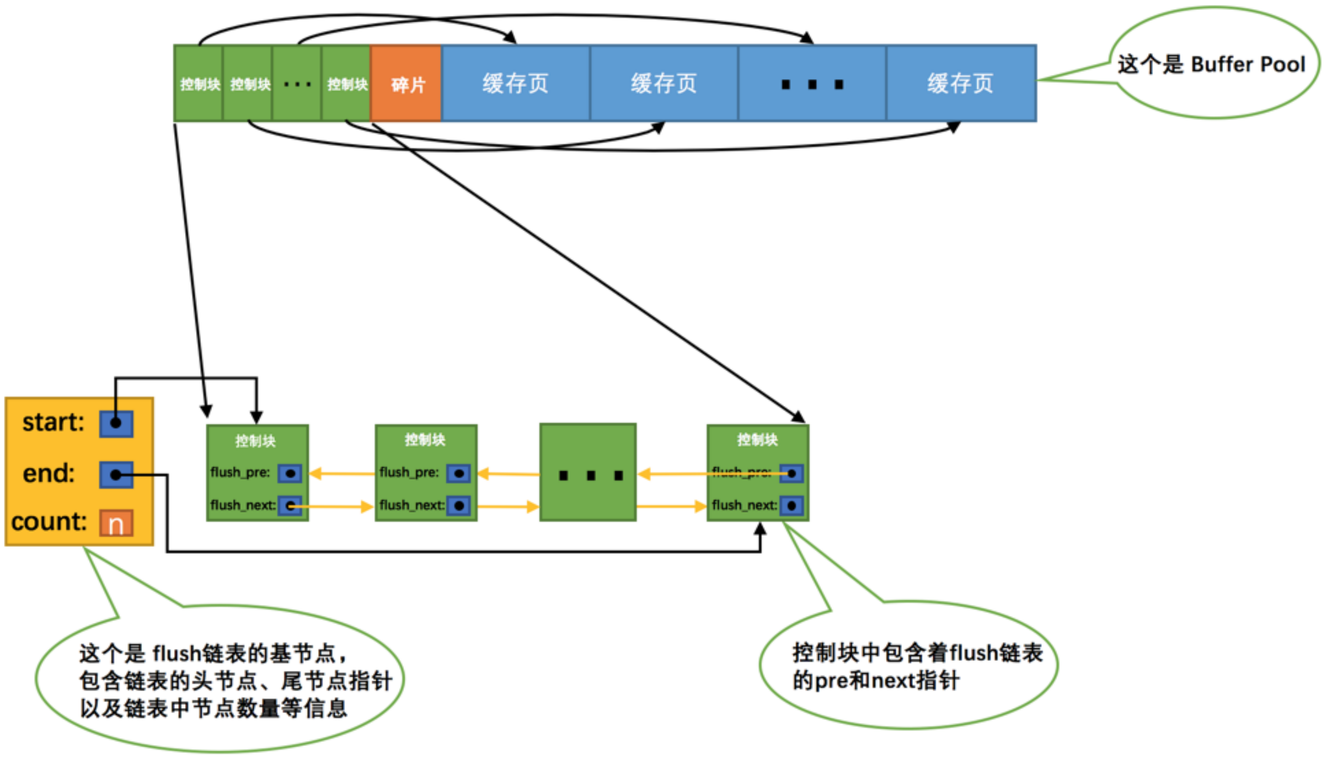
> 小贴士：
> 啥？你别告诉我你不知道哈希表是个啥？我们这个文章不是讲哈希表的，如果你不会那就去找本数据结构的书看看吧～ 啥？外头的书看不懂？别急，等我～

所以我们可以用 表空间号 + 页号 作为 key ， 缓存页 作为 value 创建一个哈希表，在需要访问某个页的数据时，先从哈希表中根据 表空间号 + 页号 看看有没有对应的缓存页，如果有，直接使用该缓存页就好，如果没有，那就从 free链表 中选一个空闲的缓存页，然后把磁盘中对应的页加载到该缓存页的位置。

18.2.5 flush链表的管理

如果我们修改了 Buffer Pool 中某个缓存页的数据，那它就和磁盘上的页不一致了，这样的缓存页也被称为 脏页 （英文名： dirty page ）。当然，最简单的做法就是每发生一次修改就立即同步到磁盘上对应的页上，但是频繁的往磁盘中写数据会严重的影响程序的性能（毕竟磁盘慢的像乌龟一样）。所以每次修改缓存页后，我们并不着急立即把修改同步到磁盘上，而是在未来的某个时间点进行同步，至于这个同步的时间点我们后边会作说明说明的，现在先不用管哈~

但是如果不立即同步到磁盘的话，那之后再同步的时候我们怎么知道 Buffer Pool 中哪些页是 脏页 ，哪些页从来没被修改过呢？总不能把所有的缓存页都同步到磁盘上吧，假如 Buffer Pool 被设置的很大，比方说 300G ，那一次性同步这么多数据岂不是要慢死！所以，我们不得不再创建一个存储脏页的链表，凡是修改过的缓存页对应的控制块都会作为一个节点加入到一个链表中，因为这个链表节点对应的缓存页都是需要被刷新到磁盘上的，所以也叫 flush链表 。链表的构造和 free链表 差不多，假设某个时间点 Buffer Pool 中的脏页数量为 n ，那么对应的 flush链表 就长这样：



18.2.6 LRU链表的管理

18.2.6.1 缓存不够的窘境

Buffer Pool 对应的内存大小毕竟是有限的，如果需要缓存的页占用的内存大小超过了 Buffer Pool 大小，也就是 free链表 中已经没有多余的空闲缓存页的时候岂不是很尴尬，发生了这样的事儿该咋办？当然是把某些旧的缓存页从 Buffer Pool 中移除，然后再把新的页放进来喽~ 那么问题来了，移除哪些缓存页呢？

为了回答这个问题，我们还需要回到我们设立 Buffer Pool 的初衷，我们就是想减少和磁盘的 IO 交互，最好每次在访问某个页的时候它都已经被缓存到 Buffer Pool 中了。假设我们一共访问了 n 次页，那么被访问的页已经在缓存中的次数除以 n 就是所谓的 缓存命中率 ，我们的期望就是让 缓存命中率 越高越好~ 从这个角度出发，回想一下我们的微信聊天列表，排在前边的都是最近很频繁使用的，排在后边的自然就是最近很少使用的，假如列表能容纳下的联系人有限，你是会把最近很频繁使用的留下还是最近很少使用的留下呢？废话，当然是留下最近很频繁使用的了~

18.2.6.2 简单的LRU链表

管理 Buffer Pool 的缓存页其实也是这个道理，当 Buffer Pool 中不再有空闲的缓存页时，就需要淘汰掉部分最近很少使用的缓存页。不过，我们怎么知道哪些缓存页最近频繁使用，哪些最近很少使用呢？呵呵，神奇的链表再一次派上了用场，我们可以再创建一个链表，由于这个链表是为了 按照最近最少使用 的原则去淘汰缓存页的，所以这个链表可以被称为 LRU链表 （LRU的英文全称：Least Recently Used）。当我们需要访问某个页时，可以这样处理 LRU链表 ：

- 如果该页不在 Buffer Pool 中，在把该页从磁盘加载到 Buffer Pool 中的缓存页时，就把该缓存页对应的 控制块 作为节点塞到链表的头部。
- 如果该页已经缓存在 Buffer Pool 中，则直接把该页对应的 控制块 移动到 LRU链表 的头部。

也就是说：只要我们使用到某个缓存页，就把该缓存页调整到 LRU链表 的头部，这样 LRU链表 尾部就是最近最少使用的缓存页喽～ 所以当 Buffer Pool 中的空闲缓存页使用完时，到 LRU链表 的尾部找些缓存页淘汰就OK啦，真简单，啧啧...

18.2.6.3 划分区域的LRU链表

高兴的太早了，上边的这个简单的 LRU链表 用了没多长时间就发现问题了，因为存在这两种比较尴尬的情况：

- 情况一： InnoDB 提供了一个看起来比较贴心的服务—— 预读 （英文名： read ahead ）。所谓 预读 ，就是 InnoDB 认为执行当前的请求可能之后会读取某些页面，就预先把它们加载到 Buffer Pool 中。根据触发方式的不同， 预读 又可以细分为下边两种：
  - 线性预读

    设计 InnoDB 的大叔提供了一个系统变量 innodb_read_ahead_threshold ，如果顺序访问了某个区（ extent ）的页面超过这个系统变量的值，就会触发一次 异步 读取下一个区中全部的页面到 Buffer Pool 的请求，注意 异步 读取意味着从磁盘中加载这些被预读的页面并不会影响到当前工作线程的正常执行。这个 innodb_read_ahead_threshold 系统变量的值默认是 56 ，我们可以在服务器启动时通过启动参数或者服务器运行过程中直接调整该系统变量的值，不过它是一个全局变量，注意使用 SET GLOBAL 命令来修改哦。

    > 小贴士：
    > InnoDB是怎么实现异步读取的呢？在Windows或者Linux平台上，可能是直接调用操作系统内核提供的AIO接口，在其它类Unix操作系统中，使用了一种模拟AIO接口的方式来实现异步读取，其实就是让别的线程去读取需要预读的页面。如果你读不懂上边这段话，那也就没必要懂了，和我们主题其实没太多关系，你只需要知道异步读取并不会影响到当前工作线程的正常执行就好了。其实这个过程涉及到操作系统如何处理IO以及多线程的问题，找本操作系统的书看看吧，什么？操作系统的书写的都很难懂？没关系，等我～

  - 随机预读

    如果 Buffer Pool 中已经缓存了某个区的13个连续的页面，不论这些页面是不是顺序读取的，都会触发一次 异步 读取本区中所有其的页面到 Buffer Pool 的请求。设计 InnoDB 的大叔同时提供了 innodb_random_read_ahead 系统变量，它的默认值为 OFF ，也就意味着 InnoDB 并不会默认开启随机预读的功能，如果我们想开启该功能，可以通过修改启动参数或者直接使用 SET GLOBAL 命令把该变量的值设置为 ON 。

  预读 本来是个好事儿，如果预读到 Buffer Pool 中的页成功的被使用到，那就可以极大的提高语句执行的效率。可是如果用不到呢？这些预读的页都会放到 LRU 链表的头部，但是如果此时 Buffer Pool 的容量不太大而且很多预读的页面都没有用到的话，这就会导致处在 LRU链表 尾部的一些缓存页会很快的被淘汰掉，也就是所谓的 劣币驱逐良币 ，会大大降低缓存命中率。
- 情况二：有的小伙伴可能会写一些需要扫描全表的查询语句（比如没有建立合适的索引或者压根儿没有WHERE子句的查询）。

扫描全表意味着什么？意味着将访问到该表所在的所有页！假设这个表中记录非常多的话，那该表会占用特别多的 页 ，当需要访问这些页时，会把它们统统都加载到 Buffer Pool 中，这也就意味着吧唧一下，Buffer Pool 中的所有页都被换了一次血，其他查询语句在执行时又得执行一次从磁盘加载到 Buffer Pool 的操作。而这种全表扫描的语句执行的频率也不高，每次执行都要把 Buffer Pool 中的缓存页换一次血，这严重的影响到其他查询对 Buffer Pool 的使用，从而大大降低了缓存命中率。

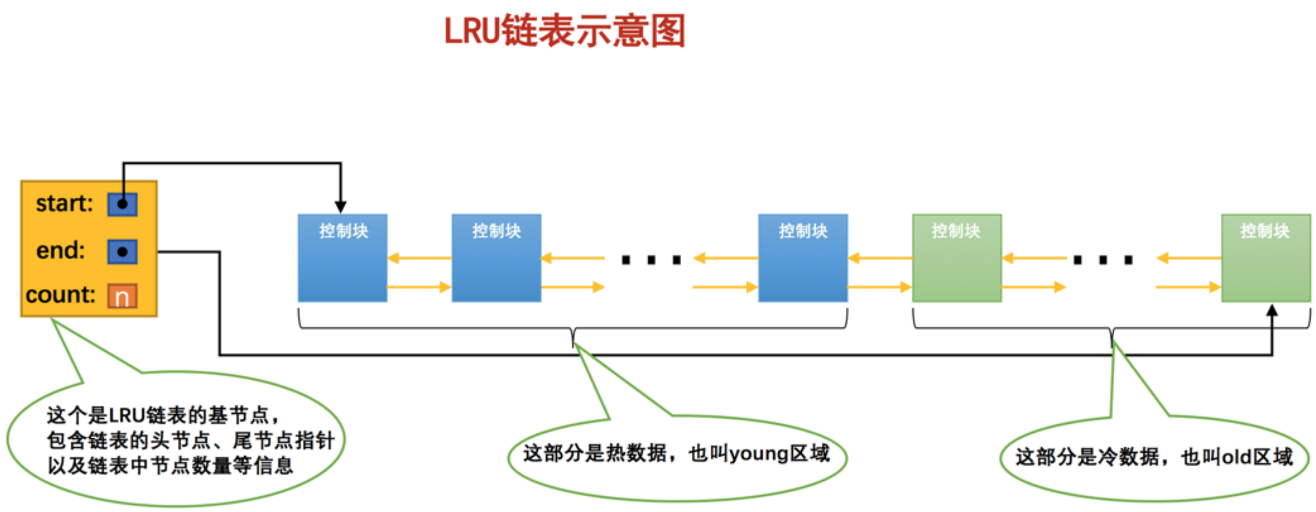
总结一下上边说的可能降低 Buffer Pool 的两种情况：

- 加载到 Buffer Pool 中的页不一定被用到。
- 如果非常多的使用频率偏低的页被同时加载到 Buffer Pool 时，可能会把那些使用频率非常高的页从 Buffer Pool 中淘汰掉。

因为有这两种情况的存在，所以设计 InnoDB 的大叔把这个 LRU链表 按照一定比例分成两截，分别是：

- 一部分存储使用频率非常高的缓存页，所以这一部分链表也叫做 热数据 ，或者称 young区域 。
- 另一部分存储使用频率不是很高的缓存页，所以这一部分链表也叫做 冷数据 ，或者称 old区域 。

为了方便大家理解，我们把示意图做了简化，各位领会精神就好：



大家要特别注意一个事儿：我们是按照某个比例将LRU链表分成两半的，不是某些节点固定是young区域的，某些节点固定是old区域的，随着程序的运行，某个节点所属的区域也可能发生变化。那这个划分成两截的比例怎么确定呢？对于 InnoDB 存储引擎来说，我们可以通过查看系统变量 innodb_old_blocks_pct 的值来确定 old 区域在 LRU链表 中所占的比例，比方说这样：

```
mysql> SHOW VARIABLES LIKE 'innodb_old_blocks_pct';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| innodb_old_blocks_pct | 37    |
+-----------------------+-------+
1 row in set (0.01 sec)
```

从结果可以看出来，默认情况下， old 区域在 LRU链表 中所占的比例是 37% ，也就是说 old 区域大约占 LRU链表 的 3/8 。这个比例我们是可以设置的，我们可以在启动时修改 innodb_old_blocks_pct 参数来控制 old 区域在 LRU链表 中所占的比例，比方说这样修改配置文件：

```
[server]
innodb_old_blocks_pct = 40
```

这样我们在启动服务器后， old 区域占 LRU链表 的比例就是 40% 。当然，如果在服务器运行期间，我们也可以修改这个系统变量的值，不过需要注意的是，这个系统变量属于 全局变量 ，一经修改，会对所有客户端生效，所以我们只能这样修改：

```
SET GLOBAL innodb_old_blocks_pct = 40;
```

有了这个被划分成 young 和 old 区域的 LRU 链表之后，设计 InnoDB 的大叔就可以针对我们上边提到的两种可能降低缓存命中率的情况进行优化了：

- 针对预读的页面可能不进行后续访问情况的优化

  设计 InnoDB 的大叔规定，当磁盘上的某个页面在初次加载到Buffer Pool中的某个缓存页时，该缓存页对应的控制块会被放到old区域的头部。这样针对预读到 Buffer Pool 却不进行后续访问的页面就会被逐渐从 old 区域逐出，而不会影响 young 区域中被使用比较频繁的缓存页。
- 针对全表扫描时，短时间内访问大量使用频率非常低的页面情况的优化

  在进行全表扫描时，虽然首次被加载到 Buffer Pool 的页被放到了 old 区域的头部，但是后续会被马上访问到，每次进行访问的时候又会把该页放到 young 区域的头部，这样仍然会把那些使用频率比较高的页面给顶下去。有同学会想：可不可以在第一次访问该页面时不将其从 old 区域移动到 young 区域的头部，后续访问时再将其移动到 young 区域的头部。回答是：行不通！因为设计 InnoDB 的大叔规定每次去页面中读取一条记录时，都算是访问一次页面，而一个页面中可能会包含很多条记录，也就是说读取完某个页面的记录就相当于访问了这个页面好多次。

  咋办？全表扫描有一个特点，那就是它的执行频率非常低，谁也不会没事儿老在那写全表扫描的语句玩，而且在执行全表扫描的过程中，即使某个页面中有很多条记录，也就是去多次访问这个页面所花费的时间也是非常少的。所以我们只需要规定，在对某个处在 old 区域的缓存页进行第一次访问时就在它对应的控制块中记录下来这个访问时间，如果后续的访问时间与第一次访问的时间在某个时间间隔内，那么该页面就不会被从old区域移动到young区域的头部，否则将它移动到young区域的头部。上述的这个间隔时间是由系统变量 innodb_old_blocks_time 控制的，你看：

  ```
  mysql> SHOW VARIABLES LIKE 'innodb_old_blocks_time';
  +------------------------+-------+
  | Variable_name          | Value |
  +------------------------+-------+
  | innodb_old_blocks_time | 1000  |
  +------------------------+-------+
  1 row in set (0.01 sec)
  ```

  这个 innodb_old_blocks_time 的默认值是 1000 ，它的单位是毫秒，也就意味着对于从磁盘上被加载到 LRU 链表的 old 区域的某个页来说，如果第一次和最后一次访问该页面的时间间隔小于 1s （很明显在一次全表扫描的过程中，多次访问一个页面中的时间不会超过 1s ），那么该页是不会被加入到 young 区域的～ 当然，像 innodb_old_blocks_pct 一样，我们也可以在服务器启动或运行时设置 innodb_old_blocks_time 的值，这里就不赘述了，你自己试试吧～ 这里需要注意的是，如果我们把 innodb_old_blocks_time 的值设置为 0 ，那么每次我们访问一个页面时就会把该页面放到 young 区域的头部。

综上所述，正是因为将 LRU 链表划分为 young 和 old 区域这两个部分，又添加了 innodb_old_blocks_time 这个系统变量，才使得预读机制和全表扫描造成的缓存命中率降低的问题得到了遏制，因为用不到的预读页面以及全表扫描的页面都只会被放到 old 区域，而不影响 young 区域中的缓存页。

18.2.6.4 更进一步优化LRU链表

LRU链表 这就说完了么？没有，早着呢～ 对于 young 区域的缓存页来说，我们每次访问一个缓存页就要把它移动到 LRU链表 的头部，这样开销是不是太大啦，毕竟在 young 区域的缓存页都是热点数据，也就是可能被经常访问的，这样频繁的对 LRU链表 进行节点移动操作是不是不太好啊？是的，为了解决这个问题其实我们还可以提出

一些优化策略，比如只有被访问的缓存页位于 young 区域的 1/4 的后边，才会被移动到 LRU链表 头部，这样就可以降低调整 LRU链表 的频率，从而提升性能（也就是说如果某个缓存页对应的节点在 young 区域的 1/4 中，再次访问该缓存页时也不会将其移动到 LRU 链表头部）。

> 小贴士：
> 我们之前介绍随机预读的时候曾说，如果Buffer Pool中有某个区的13个连续页面就会触发随机预读，这其实是不严谨的（不幸的是MySQL文档就是这么说的[摊手]），其实还要求这13个页面是非常热的页面，所谓的非常热，指的是这些页面在整个young区域的头1/4处。

还有没有什么别的针对 LRU链表 的优化措施呢？当然有啊，你要是好好学，写篇论文，写本书都不是问题，可是这毕竟是一个介绍 MySQL 基础知识的文章，再说多了篇幅就受不了了，也影响大家的阅读体验，所以适可而止，想了解更多的优化知识，自己去看源码或者更多关于 LRU 链表的知识喽～ 但是不论怎么优化，千万别忘了我们的初心：尽量高效的提高 ***Buffer Pool*** 的缓存命中率。

18.2.7 其他的一些链表

为了更好的管理 Buffer Pool 中的缓存页，除了我们上边提到的一些措施，设计 InnoDB 的大叔们还引进了其他的一些 链表 ，比如 unzip LRU链表 用于管理解压页， zip clean链表 用于管理没有被解压的压缩页， zip free数组 中每一个元素都代表一个链表，它们组成所谓的 伙伴系统 来为压缩页提供内存空间等等，反正是为了更好的管理这个 Buffer Pool 引入了各种链表或其他数据结构，具体的使用方式就不啰嗦了，大家有兴趣深究的再去找些更深的书或者直接看源代码吧，也可以直接来找我哈～

> 小贴士：
> 我们压根儿没有深入唠叨过InnoDB中的压缩页，对上边的这些链表也只是为了完整性顺便提一下，如果你看不懂千万不要抑郁，因为我压根儿就没打算向大家介绍它们。

18.2.8 刷新脏页到磁盘

后台有专门的线程每隔一段时间负责把脏页刷新到磁盘，这样可以不影响用户线程处理正常的请求。主要有两种刷新路径：

- 从 LRU链表 的冷数据中刷新一部分页面到磁盘。

  后台线程会定时从 LRU链表 尾部开始扫描一些页面，扫描的页面数量可以通过系统变量 innodb_lru_scan_depth 来指定，如果从里边儿发现脏页，会把它们刷新到磁盘。这种刷新页面的方式被称之为 BUF_FLUSH_LRU 。
- 从 flush链表 中刷新一部分页面到磁盘。

  后台线程也会定时从 flush链表 中刷新一部分页面到磁盘，刷新的速率取决于当时系统是不是很繁忙。这种刷新页面的方式被称之为 BUF_FLUSH_LIST 。

有时候后台线程刷新脏页的进度比较慢，导致用户线程在准备加载一个磁盘页到 Buffer Pool 时没有可用的缓存页，这时就会尝试看看 LRU链表 尾部有没有可以直接释放掉的未修改页面，如果没有的话会不得不将 LRU链表 尾部的一个脏页同步刷新到磁盘（和磁盘交互是很慢的，这会降低处理用户请求的速度）。这种刷新单个页面到磁盘中的刷新方式被称之为 BUF_FLUSH_SINGLE_PAGE 。

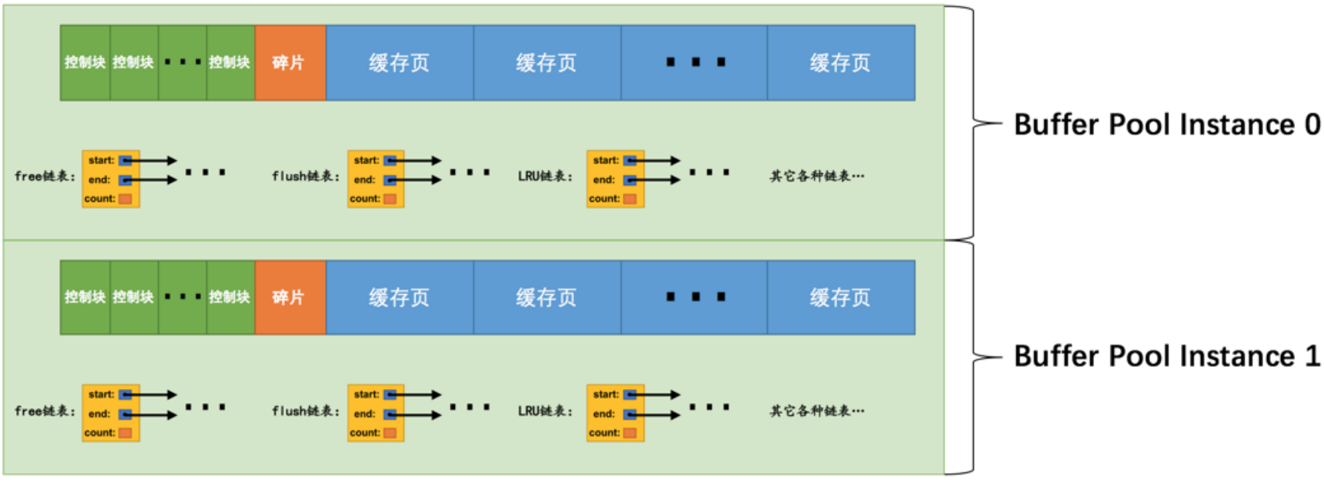
当然，有时候系统特别繁忙时，也可能出现用户线程批量的从 flush链表 中刷新脏页的情况，很显然在处理用户请求过程中去刷新脏页是一种严重降低处理速度的行为（毕竟磁盘的速度满的要死），这属于一种迫不得已的情况，不过这得放在后边唠叨 redo 日志的 checkpoint 时说了。

18.2.9 多个Buffer Pool实例

我们上边说过， Buffer Pool 本质是 InnoDB 向操作系统申请的一块连续的内存空间，在多线程环境下，访问 Buffer Pool 中的各种链表都需要加锁处理啥的，在 Buffer Pool 特别大而且多线程并发访问特别高的情况下，单一的 Buffer Pool 可能会影响请求的处理速度。所以在 Buffer Pool 特别大的时候，我们可以把它们拆分成若干个小的 Buffer Pool ，每个 Buffer Pool 都称为一个 实例 ，它们都是独立的，独立的去申请内存空间，独立的管理各种链表，独立的吧啦吧啦，所以在多线程并发访问时并不会相互影响，从而提高并发处理能力。我们可以在服务器启动的时候通过设置 innodb_buffer_pool_instances 的值来修改 Buffer Pool 实例的个数，比方说这样：

```
[server]
innodb_buffer_pool_instances = 2
```

这样就表明我们要创建2个 Buffer Pool 实例，示意图就是这样：



> 小贴士：
> 为了简便，我只把各个链表的基节点画出来了，大家应该心里清楚这些链表的节点其实就是每个缓存页对应的控制块！

那每个 Buffer Pool 实例实际占多少内存空间呢？其实使用这个公式算出来的：

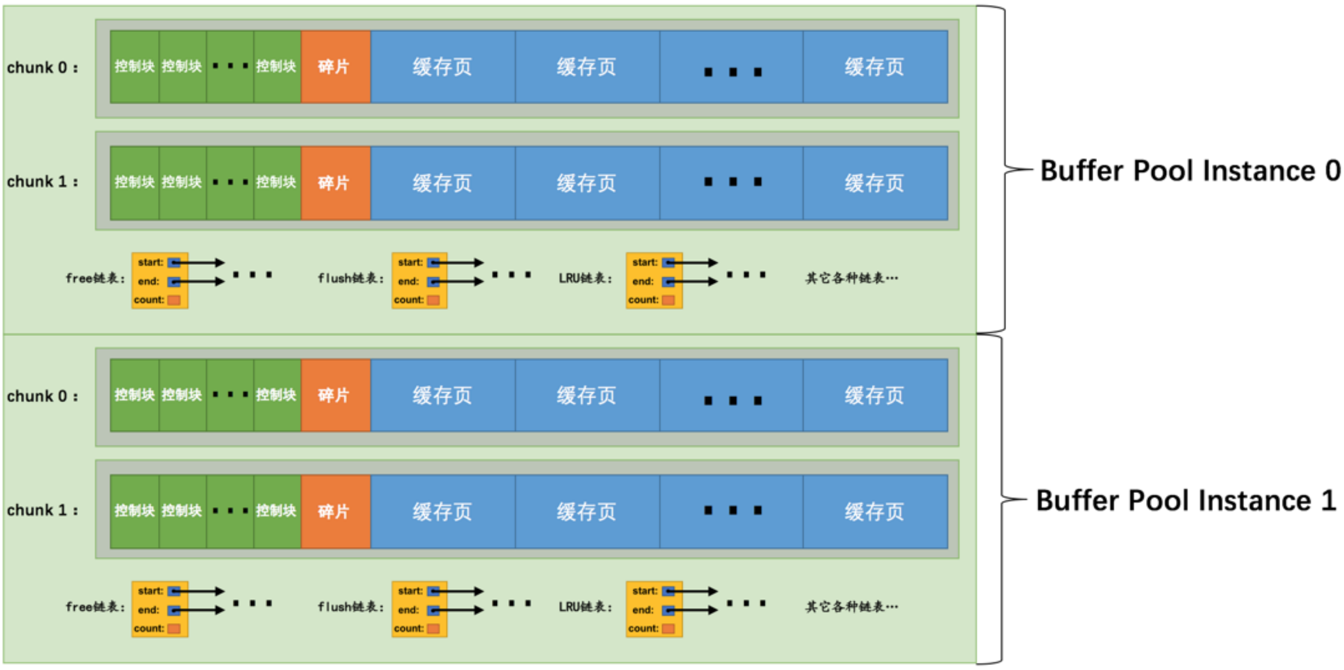
```
innodb_buffer_pool_size/innodb_buffer_pool_instances
```

也就是总共的大小除以实例的个数，结果就是每个 Buffer Pool 实例占用的大小。

不过也不是说 Buffer Pool 实例创建的越多越好，分别管理各个 Buffer Pool 也是需要性能开销的，设计 InnoDB 的大叔们规定：当innodb_buffer_pool_size的值小于1G的时候设置多个实例是无效的，InnoDB会默认把innodb_buffer_pool_instances 的值修改为1。而我们鼓励在 Buffer Pool 大小或等于1G的时候设置多个 Buffer Pool 实例。

18.2.10 innodb_buffer_pool_chunk_size

在 MySQL 5.7.5 之前， Buffer Pool 的大小只能在服务器启动时通过配置 innodb_buffer_pool_size 启动参数来调整大小，在服务器运行过程中是不允许调整该值的。不过设计 MySQL 的大叔在 5.7.5 以及之后的版本中支持了在服务器运行过程中调整 Buffer Pool 大小的功能，但是有一个问题，就是每次当我们要重新调整 Buffer Pool 大小时，都需要重新向操作系统申请一块连续的内存空间，然后将旧的 Buffer Pool 中的内容复制到这一块新空间，这是极其耗时的。所以设计 MySQL 的大叔们决定不再一次性为某个 Buffer Pool 实例向操作系统申请一大片连续的内存空间，而是以一个所谓的 chunk 为单位向操作系统申请空间。也就是说一个 Buffer Pool 实例其实是由若干个 chunk 组成的，一个 chunk 就代表一片连续的内存空间，里边儿包含了若干缓存页与其对应的控制块，画个图表示就是这样：



上图代表的 Buffer Pool 就是由2个实例组成的，每个实例中又包含2个 chunk 。

正是因为发明了这个 chunk 的概念，我们在服务器运行期间调整 Buffer Pool 的大小时就是以 chunk 为单位增加或者删除内存空间，而不需要重新向操作系统申请一片大的内存，然后进行缓存页的复制。这个所谓的 chunk 的大小是我们在启动操作 MySQL 服务器时通过 innodb_buffer_pool_chunk_size 启动参数指定的，它的默认值是 134217728 ，也就是 128M 。不过需要注意的是，innodb_buffer_pool_chunk_size的值只能在服务器启动时指定，在服务器运行过程中是不可以修改的。

> 小贴士：
> 为什么不允许在服务器运行过程中修改innodb_buffer_pool_chunk_size的值？还不是因为innodb_buffer_pool_chunk_size的值代表InnoDB向操作系统申请的一片连续的内存空间的大小，如果你在服务器运行过程中修改了该值，就意味着要重新向操作系统申请连续的内存空间并且将原先的缓存页和它们对应的控制块复制到这个新的内存空间中，这是十分耗时的操作！
> 另外，这个innodb_buffer_pool_chunk_size的值并不包含缓存页对应的控制块的内存空间大小，所以实际上InnoDB向操作系统申请连续内存空间时，每个chunk的大小要比innodb_buffer_pool_chunk_size的值大一些，约5%。

18.2.11 配置Buffer Pool时的注意事项

- innodb_buffer_pool_size 必须是 innodb_buffer_pool_chunk_size × innodb_buffer_pool_instances 的倍数（这主要是想保证每一个 Buffer Pool 实例中包含的 chunk 数量相同）。

  假设我们指定的 innodb_buffer_pool_chunk_size 的值是 128M ， innodb_buffer_pool_instances 的值是 16 ，那么这两个值的乘积就是 2G ，也就是说 innodb_buffer_pool_size 的值必须是 2G 或者 2G 的整数倍。比方说我们在启动 MySQL 服务器是这样指定启动参数的：

  ```
  mysqld --innodb-buffer-pool-size=8G --innodb-buffer-pool-instances=16
  ```

  默认的 innodb_buffer_pool_chunk_size 值是 128M ，指定的 innodb_buffer_pool_instances 的值是 16 ，所以 innodb_buffer_pool_size 的值必须是 2G 或者 2G 的整数倍，上边例子中指定的 innodb_buffer_pool_size 的值是 8G ，符合规定，所以在服务器启动完成之后我们查看一下该变量的值就是我们指定的 8G （8589934592字节）：

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| innodb_buffer_pool_size | 8589934592 |
+-------------------------+------------+
1 row in set (0.00 sec)
```

如果我们指定的 `innodb_buffer_pool_size` 大于 `2G` 并且不是 `2G` 的整数倍，那么服务器会自动的把 `innodb_buffer_pool_size` 的值调整为 `2G` 的整数倍，比方说我们在启动服务器时指定的 `innodb_buffer_pool_size` 的值是 `9G` ：

```
mysqld --innodb-buffer-pool-size=9G --innodb-buffer-pool-instances=16
```

那么服务器会自动把 `innodb_buffer_pool_size` 的值调整为 `10G` （10737418240字节），不信你看：

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+-------------+
| Variable_name           | Value       |
+-------------------------+-------------+
| innodb_buffer_pool_size | 10737418240 |
+-------------------------+-------------+
1 row in set (0.01 sec)
```

- 如果在服务器启动时， `innodb_buffer_pool_chunk_size × innodb_buffer_pool_instances` 的值已经大于 `innodb_buffer_pool_size` 的值，那么 `innodb_buffer_pool_chunk_size` 的值会被服务器自动设置为 `innodb_buffer_pool_size/innodb_buffer_pool_instances` 的值。

比方说我们在启动服务器时指定的 `innodb_buffer_pool_size` 的值为 `2G` ， `innodb_buffer_pool_instances` 的值为16， `innodb_buffer_pool_chunk_size` 的值为 `256M` ：

```
mysqld --innodb-buffer-pool-size=2G --innodb-buffer-pool-instances=16 --innodb-buffe
r-pool-chunk-size=256M
```

由于 `256M × 16 = 4G` ，而 `4G > 2G` ，所以 `innodb_buffer_pool_chunk_size` 值会被服务器改写为 `innodb_buffer_pool_size/innodb_buffer_pool_instances` 的值，也就是： `2G/16 = 128M` （134217728字节），不信你看：

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+------------+
| Variable_name           | Value      |
+-------------------------+------------+
| innodb_buffer_pool_size | 2147483648 |
+-------------------------+------------+
1 row in set (0.01 sec)

mysql> show variables like 'innodb_buffer_pool_chunk_size';
+-------------------------------+-----------+
| Variable_name                 | Value     |
+-------------------------------+-----------+
| innodb_buffer_pool_chunk_size | 134217728 |
+-------------------------------+-----------+
1 row in set (0.00 sec)
```

18.2.12 Buffer Pool中存储的其它信息

Buffer Pool 的缓存页除了用来缓存磁盘上的页面以外，还可以存储锁信息、自适应哈希索引等信息，这些内容等我们之后遇到了再详细讨论哈~

18.2.13 查看Buffer Pool的状态信息

设计 MySQL 的大叔贴心的给我们提供了 SHOW ENGINE INNODB STATUS 语句来查看关于 InnoDB 存储引擎运行过程中的一些状态信息，其中就包括 Buffer Pool 的一些信息，我们看一下（为了突出重点，我们只把输出中关于 Buffer Pool 的部分提取了出来）：

```
mysql> SHOW ENGINE INNODB STATUS\G

(...省略前边的许多状态)
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 13218349056;
Dictionary memory allocated 4014231
Buffer pool size   786432
Free buffers       8174
Database pages     710576
Old database pages 262143
Modified db pages  124941
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 6195930012, not young 78247510485
108.18 youngs/s, 226.15 non-youngs/s
Pages read 2748866728, created 29217873, written 4845680877
160.77 reads/s, 3.80 creates/s, 190.16 writes/s
Buffer pool hit rate 956 / 1000, young-making rate 30 / 1000 not 605 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 710576, unzip_LRU len: 118
I/O sum[134264]:cur[144], unzip sum[16]:cur[0]
--------------
(...省略后边的许多状态)

mysql>
```

我们来详细看一下这里边的每个值都代表什么意思：

- Total memory allocated ：代表 Buffer Pool 向操作系统申请的连续内存空间大小，包括全部控制块、缓存页、以及碎片的大小。
- Dictionary memory allocated ：为数据字典信息分配的内存空间大小，注意这个内存空间和 Buffer Pool 没啥关系，不包括在 Total memory allocated 中。
- Buffer pool size ：代表该 Buffer Pool 可以容纳多少缓存 页 ，注意，单位是 页 ！
- Free buffers ：代表当前 Buffer Pool 还有多少空闲缓存页，也就是 free链表 中还有多少个节点。
- Database pages ：代表 LRU 链表中的页的数量，包含 young 和 old 两个区域的节点数量。
- Old database pages ：代表 LRU 链表 old 区域的节点数量。
- Modified db pages ：代表脏页数量，也就是 flush链表 中节点的数量。
- Pending reads ：正在等待从磁盘上加载到 Buffer Pool 中的页面数量。

当准备从磁盘中加载某个页面时，会先为这个页面在 Buffer Pool 中分配一个缓存页以及它对应的控制块，然后把这个控制块添加到 LRU 的 old 区域的头部，但是这个时候真正的磁盘页并没有被加载进来， Pending reads 的值会跟着加1。
- Pending writes LRU ：即将从 LRU 链表中刷新到磁盘中的页面数量。
- Pending writes flush list ：即将从 flush 链表中刷新到磁盘中的页面数量。
- Pending writes single page ：即将以单个页面的形式刷新到磁盘中的页面数量。

- Pages made young ：代表 LRU 链表中曾经从 old 区域移动到 young 区域头部的节点数量。

  这里需要注意，一个节点每次只有从 old 区域移动到 young 区域头部时才会将 Pages made young 的值加1，也就是说如果该节点本来就在 young 区域，由于它符合在 young 区域1/4后边的要求，下一次访问这个页面时也会将它移动到 young 区域头部，但这个过程并不会导致 Pages made young 的值加1。
- Page made not young ：在将 innodb_old_blocks_time 设置的值大于0时，首次访问或者后续访问某个处在 old 区域的节点时由于不符合时间间隔的限制而不能将其移动到 young 区域头部时， Page made not young 的值会加1。

  这里需要注意，对于处在 young 区域的节点，如果由于它在 young 区域的1/4处而导致它没有被移动到 young 区域头部，这样的访问并不会将 Page made not young 的值加1。
- youngs/s ：代表每秒从 old 区域被移动到 young 区域头部的节点数量。
- non-youngs/s ：代表每秒由于不满足时间限制而不能从 old 区域移动到 young 区域头部的节点数量。
- Pages read 、 created 、 written ：代表读取，创建，写入了多少页。后边跟着读取、创建、写入的速率。
- Buffer pool hit rate ：表示在过去某段时间，平均访问1000次页面，有多少次该页面已经被缓存到 Buffer Pool 了。
- young-making rate ：表示在过去某段时间，平均访问1000次页面，有多少次访问使页面移动到 young 区域的头部了。

  需要大家注意的一点是，这里统计的将页面移动到 young 区域的头部次数不仅仅包含从 old 区域移动到 young 区域头部的次数，还包括从 young 区域移动到 young 区域头部的次数（访问某个 young 区域的节点，只要该节点在 young 区域的1/4处往后，就会把它移动到 young 区域的头部）。
- not (young-making rate) ：表示在过去某段时间，平均访问1000次页面，有多少次访问没有使页面移动到 young 区域的头部。

  需要大家注意的一点是，这里统计的没有将页面移动到 young 区域的头部次数不仅仅包含因为设置了 innodb_old_blocks_time 系统变量而导致访问了 old 区域中的节点但没把它们移动到 young 区域的次数，还包含因为该节点在 young 区域的前1/4处而没有被移动到 young 区域头部的次数。
- LRU len ：代表 LRU链表 中节点的数量。
- unzip_LRU ：代表 unzip_LRU链表 中节点的数量（由于我们没有具体唠叨过这个链表，现在可以忽略它的值）。
- I/O sum ：最近50s读取磁盘页的总数。
- I/O cur ：现在正在读取的磁盘页数量。
- I/O unzip sum ：最近50s解压的页面数量。
- I/O unzip cur ：正在解压的页面数量。

18.3 总结

1. 磁盘太慢，用内存作为缓存很有必要。
2. Buffer Pool 本质上是 InnoDB 向操作系统申请的一段连续的内存空间，可以通过 innodb_buffer_pool_size 来调整它的大小。
3. Buffer Pool 向操作系统申请的连续内存由控制块和缓存页组成，每个控制块和缓存页都是一一对应的，在填充足够多的控制块和缓存页的组合后， Buffer Pool 剩余的空间可能产生不够填充一组控制块和缓存页，这部分空间不能被使用，也被称为 碎片 。
4. InnoDB 使用了许多 链表 来管理 Buffer Pool 。

5. free链表 中每一个节点都代表一个空闲的缓存页，在将磁盘中的页加载到 Buffer Pool 时，会从 free链表 中寻找空闲的缓存页。
6. 为了快速定位某个页是否被加载到 Buffer Pool ，使用 表空间号 + 页号 作为 key ，缓存页作为 value ，建立哈希表。
7. 在 Buffer Pool 中被修改的页称为 脏页 ，脏页并不是立即刷新，而是被加入到 flush链表 中，待之后的某个时刻同步到磁盘上。
8. LRU链表 分为 young 和 old 两个区域，可以通过 innodb_old_blocks_pct 来调节 old 区域所占的比例。首次从磁盘上加载到 Buffer Pool 的页会被放到 old 区域的头部，在 innodb_old_blocks_time 间隔时间内访问该页不会把它移动到 young 区域头部。在 Buffer Pool 没有可用的空闲缓存页时，会首先淘汰掉 old 区域的一些页。
9. 我们可以通过指定 innodb_buffer_pool_instances 来控制 Buffer Pool 实例的个数，每个 Buffer Pool 实例中都有各自独立的链表，互不干扰。
10. 自 MySQL 5.7.5 版本之后，可以在服务器运行过程中调整 Buffer Pool 大小。每个 Buffer Pool 实例由若干个 chunk 组成，每个 chunk 的大小可以在服务器启动时通过启动参数调整。
11. 可以用下边的命令查看 Buffer Pool 的状态信息：

```
SHOW ENGINE INNODB STATUS\G
```

19 第19章 从猫爷被杀说起-事务简介

标签： MySQL是怎样运行的

19.1 事务的起源

对于大部分程序员来说，他们的任务就是把现实世界的业务场景映射到数据库世界。比如银行为了存储人们的账户信息会建立一个 account 表：

```
CREATE TABLE account (
    id INT NOT NULL AUTO_INCREMENT COMMENT '自增id',
    name VARCHAR(100) COMMENT '客户名称',
    balance INT COMMENT '余额',
    PRIMARY KEY (id)
) Engine=InnoDB CHARSET=utf8;
```

狗哥和猫爷是一对好基友，他们都到银行开一个账户，他们在现实世界中拥有的资产就会体现在数据库世界的 account 表中。比如现在狗哥有 11 元，猫爷只有 2 元，那么现实中的这个情况映射到数据库的 account 表就是这样：

```
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |      11 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
```

在某个特定的时刻，狗哥猫爷这些家伙在银行所拥有的资产是一个特定的值，这些特定的值也可以被描述为账户在这个特定的时刻现实世界的一个状态。随着时间的流逝，狗哥和猫爷可能陆续进行向账户中存钱、取钱或者向别人转账等操作，这样他们账户中的余额就可能发生变动，每一个操作都相当于现实世界中账户的一次状态转换。数据库世界作为现实世界的一个映射，自然也要进行相应的变动。不变不知道，一变吓一跳，现实世界中一些看似很简单的状态转换，映射到数据库世界却不是那么容易的。比方说有一次猫爷在赌场赌博输了钱，急忙打

电话给狗哥要借10块钱，不然那些看场子的就会把自己剁了。现实世界中的狗哥走向了ATM机，输入了猫爷的账号以及10元的转账金额，然后按下确认，狗哥就拔卡走人了。对于数据库世界来说，相当于执行了下边这两条语句：

```
UPDATE account SET balance = balance - 10 WHERE id = 1;
UPDATE account SET balance = balance + 10 WHERE id = 2;
```

但是这里头有个问题，上述两条语句只执行了一条时忽然服务器断电了咋办？把狗哥的钱扣了，但是没给猫爷转过去，那猫爷还是逃脱不了被砍死的噩运～ 即使对于单独的一条语句，我们前边唠叨 `Buffer Pool` 时也说过，在对某个页面进行读写访问时，都会先把这个页面加载到 `Buffer Pool` 中，之后如果修改了某个页面，也不会立即把修改同步到磁盘，而只是把这个修改了的页面加到 `Buffer Pool` 的 `flush链表` 中，在之后的某个时间点才会刷新到磁盘。如果在将修改过的页刷新到磁盘之前系统崩溃了那岂不是猫爷还是要被砍死？或者在刷新磁盘的过程中（只刷新部分数据到磁盘上）系统奔溃了猫爷也会被砍死？

怎么才能保证让可怜的猫爷不被砍死呢？其实再仔细想想，我们只是想让某些数据库操作符合现实世界中状态转换的规则而已，设计数据库的大叔们仔细盘算了盘算，现实世界中状态转换的规则有好几条，待我们慢慢道来。

19.1.1 原子性（Atomicity）

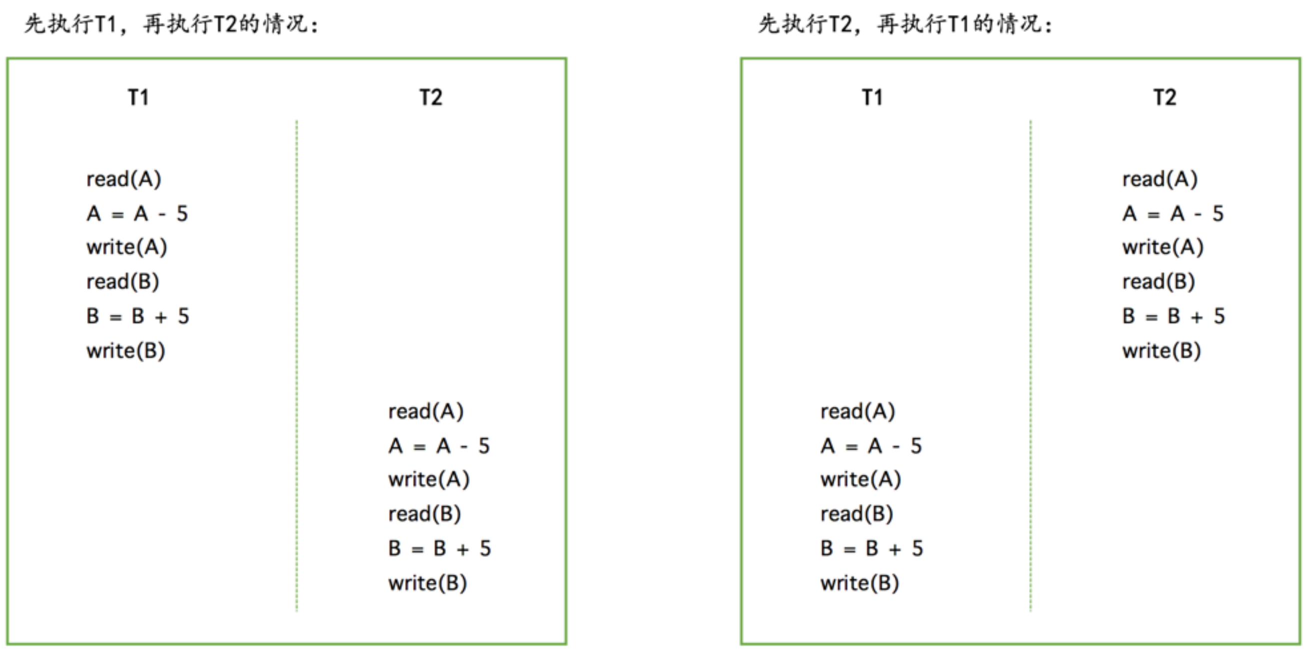
现实世界中转账操作是一个不可分割的操作，也就是说要么压根儿就没转，要么转账成功，不能存在中间的状态，也就是转了一半的这种情况。设计数据库的大叔们把这种要么全做，要么全不做的规则称之为 `原子性` 。但是在现实世界中的一个不可分割的操作却可能对应着数据库世界若干条不同的操作，数据库中的一条操作也可能被分解成若干个步骤（比如先修改缓存页，之后再刷新到磁盘等），最要命的是在任何一个可能的时间都可能发生意想不到的错误（可能是数据库本身的错误，或者是操作系统错误，甚至是直接断电之类的）而使操作执行不下去，所以猫爷可能会被砍死。为了保证在数据库世界中某些操作的原子性，设计数据库的大叔需要费一些心机来保证如果在执行操作的过程中发生了错误，把已经做了的操作恢复成没执行之前的样子，这也是我们后边章节要仔细唠叨的内容。

19.1.2 隔离性（Isolation）

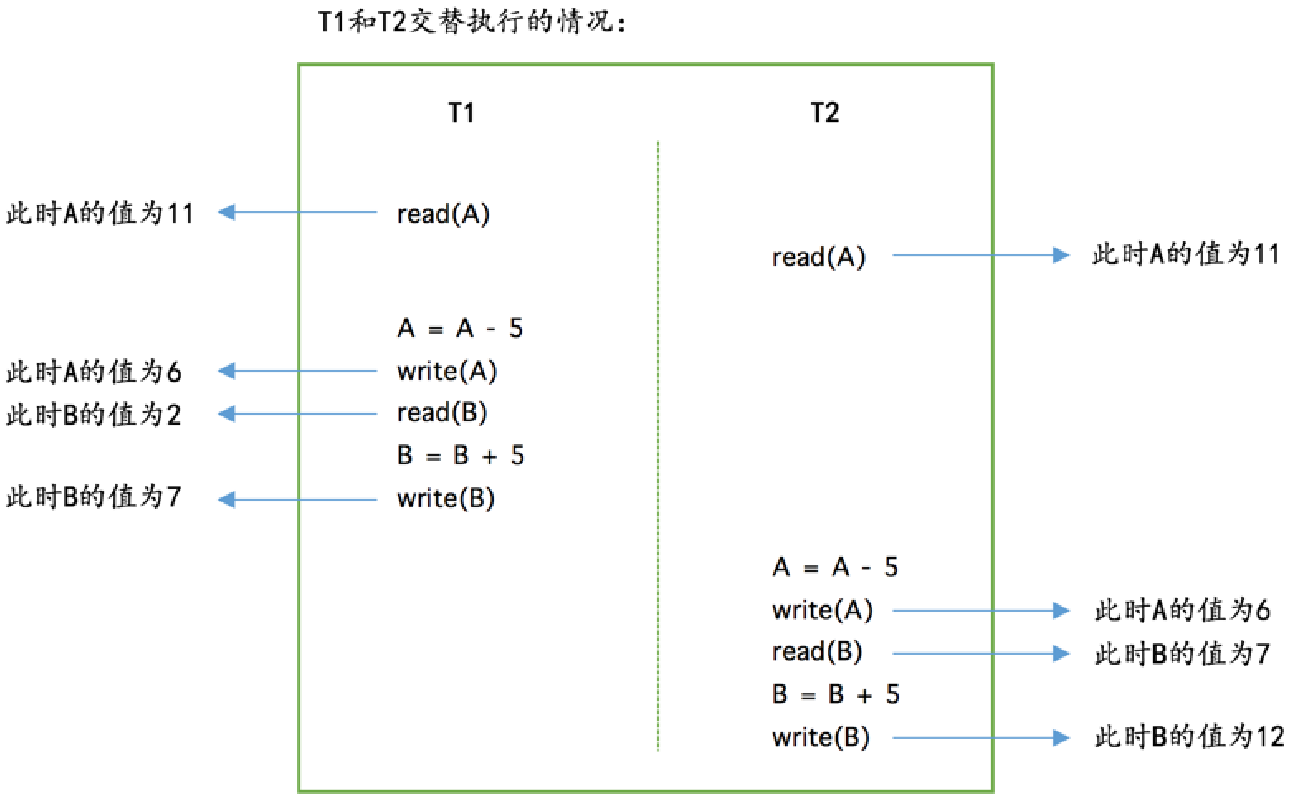
现实世界中的两次状态转换应该是互不影响的，比如说狗哥向猫爷同时进行的两次金额为5元的转账（假设可以在两个ATM机上同时操作）。那么最后狗哥的账户里肯定会少10元，猫爷的账户里肯定多了10元。但是到对应的数据库世界中，事情又变的复杂了一些。为了简化问题，我们粗略的假设狗哥向猫爷转账5元的过程是由下边几个步骤组成的：

- 步骤一：读取狗哥账户的余额到变量A中，这一步骤简写为 `read(A)` 。
- 步骤二：将狗哥账户的余额减去转账金额，这一步骤简写为 `A = A - 5` 。
- 步骤三：将狗哥账户修改过的余额写到磁盘里，这一步骤简写为 `write(A)` 。
- 步骤四：读取猫爷账户的余额到变量B，这一步骤简写为 `read(B)` 。
- 步骤五：将猫爷账户的余额加上转账金额，这一步骤简写为 `B = B + 5` 。
- 步骤六：将猫爷账户修改过的余额写到磁盘里，这一步骤简写为 `write(B)` 。

我们将狗哥向猫爷同时进行的两次转账操作分别称为 `T1` 和 `T2` ，在现实世界中 `T1` 和 `T2` 是应该没有关系的，可以先执行完 `T1` ，再执行 `T2` ，或者先执行完 `T2` ，再执行 `T1` ，对应的数据库操作就像这样：



但是很不幸，真实的数据库中 T1 和 T2 的操作可能交替执行，比如这样：



如果按照上图中的执行顺序来进行两次转账的话，最终狗哥的账户里还剩 6 元钱，相当于只扣了5元钱，但是猫爷的账户里却成了 12 元钱，相当于多了10元钱，这银行岂不是要亏死了？

所以对于现实世界中状态转换对应的某些数据库操作来说，不仅要保证这些操作以 原子性 的方式执行完成，而且要保证其它的状态转换不会影响到本次状态转换，这个规则被称之为 隔离性 。这时设计数据库的大叔们就需要采取一些措施来让访问相同数据（上例中的A账户和B账户）的不同状态转换（上例中的 T1 和 T2 ）对应的数据库操作的执行顺序有一定规律，这也是我们后边章节要仔细唠叨的内容。

19.1.3 一致性（Consistency）

我们生活的这个世界存在着形形色色的约束，比如身份证号不能重复，性别只能是男或者女，高考的分数只能在0～750之间，人民币面值最大只能是100（现在是2019年），红绿灯只有3种颜色，房价不能为负的，学生要听老师话，吧啦吧啦有点儿扯远了～ 只有符合这些约束的数据才是有效的，比如有个小孩儿跟你说他高考考了1000分，你一听就知道他胡扯呢。数据库世界只是现实世界的一个映射，现实世界中存在的约束当然也要在数据库世界中有所体现。如果数据库中的数据全部符合现实世界中的约束（all defined rules），我们说这些数据就是一致的，或者说符合 `一致性` 的。

如何保证数据库中数据的一致性（就是符合所有现实世界的约束）呢？这其实靠两方面的努力：

- 数据库本身能为我们保证一部分一致性需求（就是数据库自身可以保证一部分现实世界的约束永远有效）。

  我们知道 `MySQL` 数据库可以为表建立主键、唯一索引、外键、声明某个列为 `NOT NULL` 来拒绝 `NULL` 值的插入。比如说当我们对某个列建立唯一索引时，如果插入某条记录时该列的值重复了，那么 `MySQL` 就会报错并且拒绝插入。除了这些我们已经非常熟悉的保证一致性的功能， `MySQL` 还支持 `CHECK` 语法来自定义约束，比如这样：

  ```
  CREATE TABLE account (
      id INT NOT NULL AUTO_INCREMENT COMMENT '自增id',
      name VARCHAR(100) COMMENT '客户名称',
      balance INT COMMENT '余额',
      PRIMARY KEY (id),
      CHECK (balance >= 0)
  );
  ```

  上述例子中的 `CHECK` 语句本意是想规定 `balance` 列不能存储小于0的数字，对应的现实世界的意思就是银行账户余额不能小于0。但是很遗憾，MySQL仅仅支持CHECK语法，但实际上并没有一点卵用，也就是说即使我们使用上述带有 `CHECK` 子句的建表语句来创建 `account` 表，那么在后续插入或更新记录时， `MySQL` 并不会去检查 `CHECK` 子句中的约束是否成立。

  > 小贴士：
  > 其它的一些数据库，比如SQL Server或者Oracle支持的CHECK语法是有实实在在的作用的，每次进行插入或更新记录之前都会检查一下数据是否符合CHECK子句中指定的约束条件是否成立，如果不成立的话就会拒绝插入或更新。

  虽然 `CHECK` 子句对一致性检查没什么卵用，但是我们还是可以通过定义触发器的方式来自定义一些约束条件以保证数据库中数据的一致性。

  > 小贴士：
  > 触发器是MySQL基础内容中的知识，本书是一本MySQL进阶的书籍，如果你不了解触发器，那恐怕要找本基础内容的书籍来看看了。

- 更多的一致性需求需要靠写业务代码的程序员自己保证。

  为建立现实世界和数据库世界的对应关系，理论上应该把现实世界中的所有约束都反应到数据库世界中，但是很不幸，在更改数据库数据时进行一致性检查是一个耗费性能的工作，比方说我们为 `account` 表建立了一个触发器，每当插入或者更新记录时都会校验一下 `balance` 列的值是不是大于0，这就会影响到插入或更新的速度。仅仅是校验一行记录符不符合一致性需求倒也不是什么大问题，有的一致性需求简直变态，比方说银行会建立一张代表账单的表，里边儿记录了每个账户的每笔交易，每一笔交易完成后，都需要保证整个系统的余额等于所有账户的收入减去所有账户的支出。如果在数据库层面实现这个一致性需求的话，每次发生交易时，都需要将所有的收入加起来减去所有的支出，再将所有的账户余额加起来，看看两个值相不相等。这不是搞笑呢么，如果账单表里有几亿条记录，光是这个校验的过程可能就要跑好几个小时，也就是说你在煎饼摊买个煎饼，使用银行卡付款之后要等好几个小时才能提示付款成功，这样的性能代价是完全承受不起的。

现实生活中复杂的一致性需求比比皆是，而由于性能问题把一致性需求交给数据库去解决这是不现实的，所以这个锅就甩给了业务端程序员。比方说我们的 account 表，我们也可以不建立触发器，只要编写业务的程序员在自己的业务代码里判断一下，当某个操作会将 balance 列的值更新为小于0的值时，就不执行该操作就好了嘛！

我们前边唠叨的 原子性 和 隔离性 都会对 一致性 产生影响，比如我们现实世界中转账操作完成后，有一个 一致性 需求就是参与转账的账户的总的余额是不变的。如果数据库不遵循 原子性 要求，也就是转了一半就不转了，也就是说给狗哥扣了钱而没给猫爷转过去，那最后就是不符合一致性需求的；类似的，如果数据库不遵循 隔离性 要求，就像我们前边唠叨 隔离性 时举的例子中所说的，最终狗哥账户中扣的钱和猫爷账户中涨的钱可能就不一样了，也就是说不符合 一致性 需求了。所以说，数据库某些操作的原子性和隔离性都是保证一致性的一种手段，在操作执行完成后保证符合所有既定的约束则是一种结果。那满足 原子性 和 隔离性 的操作一定就满足 一致性 么？那倒也不一定，比如说狗哥要转账20元给猫爷，虽然在满足 原子性 和 隔离性 ，但转账完成了之后狗哥的账户的余额就成负的了，这显然是不满足 一致性 的。那不满足 原子性 和 隔离性 的操作就一定不满足 一致性 么？这也不一定，只要最后的结果符合所有现实世界中的约束，那么就是符合 一致性 的。

19.1.4 持久性（Durability）

当现实世界的一个状态转换完成后，这个转换的结果将永久的保留，这个规则被设计数据库的大叔们称为 持久性 。比方说狗哥向猫爷转账，当ATM机提示转账成功了，就意味着这次账户的状态转换完成了，狗哥就可以拔卡走人了。如果当狗哥走掉之后，银行又把这次转账操作给撤销掉，恢复到没转账之前的样子，那猫爷不就惨了，又得被砍死了，所以这个 持久性 是非常重要的。

当把现实世界的状态转换映射到数据库世界时， 持久性 意味着该转换对应的数据库操作所修改的数据都应该在磁盘上保留下来，不论之后发生了什么事故，本次转换造成的影响都不应该被丢失掉（要不然猫爷还是会被砍死）。

19.2 事务的概念

为了方便大家记住我们上边唠叨的现实世界状态转换过程中需要遵守的4个特性，我们把 原子性 （ Atomicity ）、 隔离性 （ Isolation ）、 一致性 （ Consistency ）和 持久性 （ Durability ）这四个词对应的英文单词首字母提取出来就是 A 、 I 、 C 、 D ，稍微变换一下顺序可以组成一个完整的英文单词： ACID 。想必大家都是学过初高中英语的， ACID 是英文 酸 的意思，以后我们提到 ACID 这个词儿，大家就应该想到原子性、一致性、隔离性、持久性这几个规则。另外，设计数据库的大叔为了方便起见，把需要保证 原子性 、 隔离性 、 一致性 和 持久性 的一个或多个数据库操作称之为一个 事务 （英文名是： transaction ）。

我们现在知道 事务 是一个抽象的概念，它其实对应着一个或多个数据库操作，设计数据库的大叔根据这些操作所执行的不同阶段把 事务 大致上划分成了这么几个状态：

- 活动的（active）

  事务对应的数据库操作正在执行过程中时，我们就说该事务处在 活动的 状态。
- 部分提交的（partially committed）

  当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并没有刷新到磁盘时，我们就说该事务处在 部分提交的 状态。
- 失败的（failed）

  当事务处在 活动的 或者 部分提交的 状态时，可能遇到了某些错误（数据库自身的错误、操作系统错误或者直接断电等）而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在 失败的 状态。
- 中止的（aborted）

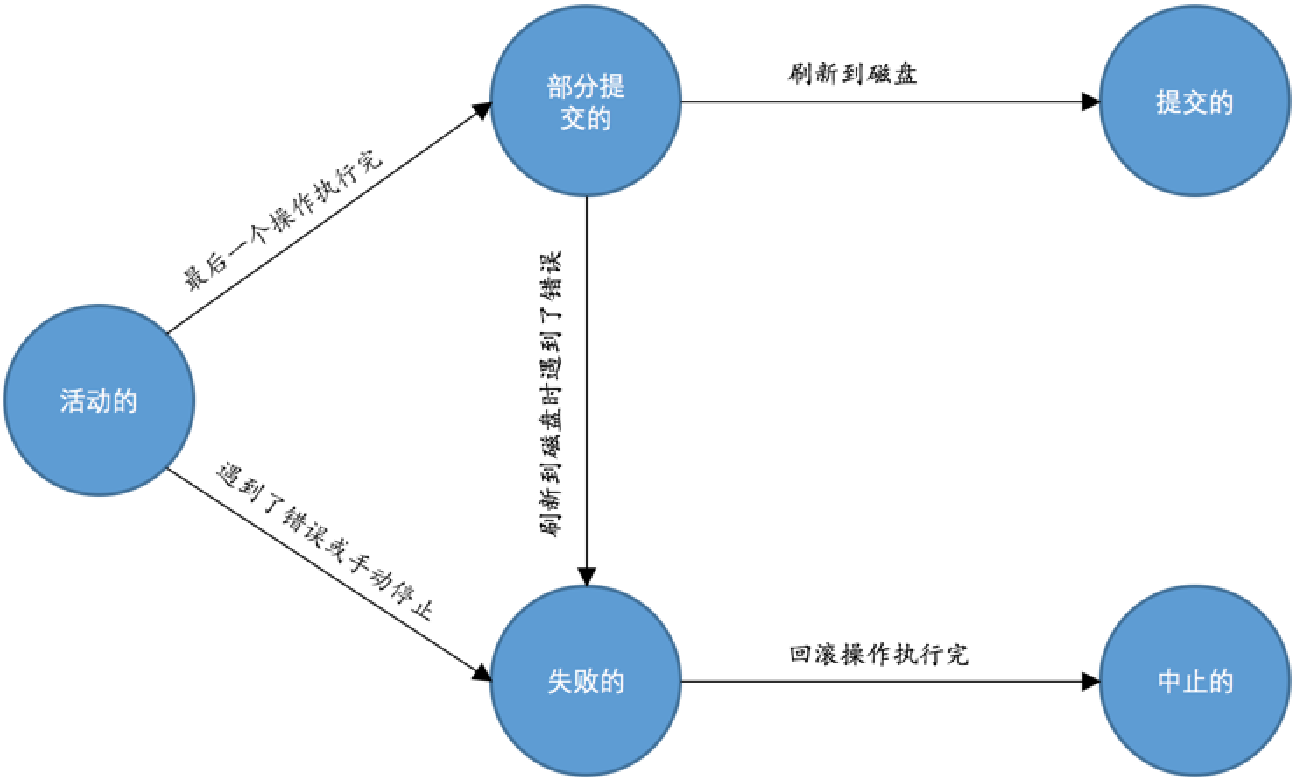
  如果事务执行了半截而变为 失败的 状态，比如我们前边唠叨的狗哥向猫爷转账的事务，当狗哥账户的钱被扣除，但是猫爷账户的钱没有增加时遇到了错误，从而当前事务处在了 失败的 状态，那么就需要把已经修改的狗哥账户余额调整为未转账之前的金额，换句话说，就是要撤销失败事务对当前数据库造成的影响。书

面一点的话，我们把这个撤销的过程称之为 回滚 。当 回滚 操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了 中止的 状态。

- 提交的（committed）

  当一个处在 部分提交的 状态的事务将修改过的数据都同步到磁盘上之后，我们就可以说该事务处在了 提交的 状态。

随着事务对应的数据库操作执行到不同阶段，事务的状态也在不断变化，一个基本的状态转换图如下所示：



从图中大家也可以看出了，只有当事务处于提交的或者中止的状态时，一个事务的生命周期才算是结束了。对于已经提交的事务来说，该事务对数据库所做的修改将永久生效，对于处于中止状态的事务，该事务对数据库所做的所有修改都会被回滚到没执行该事务之前的状态。

> 小贴士：
> 此贴士处纯属扯犊子，与正文没啥关系，纯属吐槽。大家知道我们的计算机术语基本上全是从英文翻译成中文的，事务的英文是transaction，英文直译就是交易，买卖的意思，交易就是买的人付钱，卖的人交货，不能付了钱不交货，交了货不付钱把，所以交易本身就是一种不可分割的操作。不知道是哪位大神把transaction翻译成了事务（我想估计是他们也想不出什么更好的词儿，只能随便找一个了），事务这个词儿完全没有交易、买卖的意思，所以大家理解起来也会比较困难，外国人理解transaction可能更好理解一点吧～

19.3 MySQL中事务的语法

我们说 事务 的本质其实只是一系列数据库操作，只不过这些数据库操作符合 ACID 特性而已，那么 MySQL 中如何将某些操作放到一个事务里去执行的呢？我们下边就来重点唠叨唠叨。

19.3.1 开启事务

我们可以使用下边两种语句之一来开启一个事务：

- BEGIN [WORK];

BEGIN 语句代表开启一个事务，后边的单词 WORK 可有可无。开启事务后，就可以继续写若干条语句，这些语句都属于刚刚开启的这个事务。

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> 加入事务的语句...
```

- START TRANSACTION;

START TRANSACTION 语句和 BEGIN 语句有着相同的功效，都标志着开启一个事务，比如这样：

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> 加入事务的语句...
```

不过比 BEGIN 语句牛逼一点儿的是，可以在 START TRANSACTION 语句后边跟随几个 修饰符 ，就是它们几个：

- READ ONLY ：标识当前事务是一个只读事务，也就是属于该事务的数据库操作只能读取数据，而不能修改数据。

  > 小贴士：
  > 其实只读事务中只是不允许修改那些其他事务也能访问到的表中的数据，对于临时表来说（我们使用CREATE TMEPORARY TABLE创建的表），由于它们只能在当前会话中可见，所以只读事务其实也是可以对临时表进行增、删、改操作的。

- READ WRITE ：标识当前事务是一个读写事务，也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。
- WITH CONSISTENT SNAPSHOT ：启动一致性读（先不用关心啥是个一致性读，后边的章节才会唠叨）。

  比如我们想开启一个只读事务的话，直接把 READ ONLY 这个修饰符加在 START TRANSACTION 语句后边就好，比如这样：

  ```
  START TRANSACTION READ ONLY;
  ```

  如果我们想在 START TRANSACTION 后边跟随多个 修饰符 的话，可以使用逗号将 修饰符 分开，比如开启一个只读事务和一致性读，就可以这样写：

  ```
  START TRANSACTION READ ONLY, WITH CONSISTENT SNAPSHOT;
  ```

  或者开启一个读写事务和一致性读，就可以这样写：

  ```
  START TRANSACTION READ WRITE, WITH CONSISTENT SNAPSHOT
  ```

  不过这里需要大家注意的一点是， READ ONLY 和 READ WRITE 是用来设置所谓的事务 访问模式 的，就是以只读还是读写的方式来访问数据库中的数据，一个事务的访问模式不能同时既设置为 只读 的也设置为 读写 的，所以我们不能同时把 READ ONLY 和 READ WRITE 放到 START TRANSACTION 语句后边。另外，如果我们不显式指定事务的访问模式，那么该事务的访问模式就是 读写 模式。

19.3.2 提交事务

开启事务之后就可以继续写需要放到该事务中的语句了，当最后一条语句写完了之后，我们就可以提交该事务了，提交的语句也很简单：

```
COMMIT [WORK]
```

COMMIT 语句就代表提交一个事务，后边的 WORK 可有可无。比如我们上边说狗哥给猫爷转10元钱其实对应 MySQL 中的两条语句，我们就可以把这两条语句放到一个事务中，完整的过程就是这样：

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.02 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE account SET balance = balance + 10 WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

19.3.3 手动中止事务

如果我们写了几条语句之后发现上边的某条语句写错了，我们可以手动的使用下边这个语句来将数据库恢复到事务执行之前的样子：

```
ROLLBACK [WORK]
```

ROLLBACK 语句就代表中止并回滚一个事务，后边的 WORK 可有可无类似的。比如我们在写狗哥给猫爷转账10元钱对应的 MySQL 语句时，先给狗哥扣了10元，然后一时大意只给猫爷账户上增加了1元，此时就可以使用 ROLLBACK 语句进行回滚，完整的过程就是这样：

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> UPDATE account SET balance = balance + 1 WHERE id = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

这里需要强调一下， ROLLBACK 语句是我们程序员手动的去回滚事务时才去使用的，如果事务在执行过程中遇到了某些错误而无法继续执行的话，事务自身会自动的回滚。

> 小贴士：
> 我们这里所说的开启、提交、中止事务的语法只是针对使用黑框框时通过mysql客户端程序与服务器进行交互时控制事务的语法，如果大家使用的是别的客户端程序，比如JDBC之类的，那需要参考相应的文档来看看如何控制事务。

19.3.4 支持事务的存储引擎

MySQL 中并不是所有存储引擎都支持事务的功能，目前只有 InnoDB 和 NDB 存储引擎支持（NDB存储引擎不是我们的重点），如果某个事务中包含了修改使用不支持事务的存储引擎的表，那么对该使用不支持事务的存储引擎的表所做的修改将无法进行回滚。比方说我们有两个表， tbl1 使用支持事务的存储引擎 InnoDB ， tbl2 使用不支持事务的存储引擎 MyISAM ，它们的建表语句如下所示：

```
CREATE TABLE tbl1 (
    i int
) engine=InnoDB;

CREATE TABLE tbl2 (
    i int
) ENGINE=MyISAM;
```

我们看看先开启一个事务，写一条插入语句后再回滚该事务， tbl1 和 tbl2 的表现有什么不同：

```
mysql> SELECT * FROM tbl1;
Empty set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tbl1 VALUES(1);
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tbl1;
Empty set (0.00 sec)
```

可以看到，对于使用支持事务的存储引擎的 tbl1 表来说，我们在插入一条记录再回滚后， tbl1 就恢复到没有插入记录时的状态了。再看看 tbl2 表的表现：

```
mysql> SELECT * FROM tbl2;
Empty set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tbl2 VALUES(1);
Query OK, 1 row affected (0.00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SELECT * FROM tbl2;
+------+
| i    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

可以看到，虽然我们使用了 ROLLBACK 语句来回滚事务，但是插入的那条记录还是留在了 tbl2 表中。

19.3.5 自动提交

MySQL 中有一个系统变量 autocommit :

```
mysql> SHOW VARIABLES LIKE 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
1 row in set (0.01 sec)
```

可以看到它的默认值为 ON ，也就是说默认情况下，如果我们不显式的使用 START TRANSACTION 或者 BEGIN 语句开启一个事务，那么每一条语句都算是一个独立的事务，这种特性称之为事务的 自动提交 。假如我们在狗哥向猫爷转账10元时不以 START TRANSACTION 或者 BEGIN 语句显式的开启一个事务，那么下边这两条语句就相当于放到两个独立的事务中去执行：

```
UPDATE account SET balance = balance - 10 WHERE id = 1;
UPDATE account SET balance = balance + 10 WHERE id = 2;
```

当然，如果我们想关闭这种 自动提交 的功能，可以使用下边两种方法之一：

- 显式的的使用 START TRANSACTION 或者 BEGIN 语句开启一个事务。

  这样在本次事务提交或者回滚前会暂时关闭掉自动提交的功能。

- 把系统变量 autocommit 的值设置为 OFF ，就像这样：

  ```
  SET autocommit = OFF;
  ```

  这样的话，我们写入的多条语句就算是属于同一个事务了，直到我们显式的写出 COMMIT 语句来把这个事务提交掉，或者显式的写出 ROLLBACK 语句来把这个事务回滚掉。

19.3.6 隐式提交

当我们使用 START TRANSACTION 或者 BEGIN 语句开启了一个事务，或者把系统变量 autocommit 的值设置为 OFF 时，事务就不会进行 自动提交 ，但是如果我们输入了某些语句之后就会 悄悄的 提交掉，就像我们输入了 COMMIT 语句了一样，这种因为某些特殊的语句而导致事务提交的情况称为 隐式提交 ，这些会导致事务隐式提交的语句包括：

- 定义或修改数据库对象的数据定义语言（Data definition language，缩写为： DDL ）。

  所谓的数据库对象，指的就是 数据库 、 表 、 视图 、 存储过程 等等这些东西。当我们使用 CREATE 、 ALTER 、 DROP 等语句去修改这些所谓的数据库对象时，就会隐式的提交前边语句所属于的事务，就像这样：

  ```
  BEGIN;

  SELECT ... # 事务中的一条语句
  UPDATE ... # 事务中的一条语句
  ... # 事务中的其它语句

  CREATE TABLE ... # 此语句会隐式的提交前边语句所属于的事务
  ```

- 隐式使用或修改 mysql 数据库中的表

当我们使用 ALTER USER 、 CREATE USER 、 DROP USER 、 GRANT 、 RENAME USER 、 REVOKE 、 SET PASSWORD 等语句时也会隐式的提交前边语句所属于的事务。

- 事务控制或关于锁定的语句

  当我们在一个事务还没提交或者回滚时就又使用 START TRANSACTION 或者 BEGIN 语句开启了另一个事务时，会隐式的提交上一个事务，比如这样：

  ```
  BEGIN;

  SELECT ... # 事务中的一条语句
  UPDATE ... # 事务中的一条语句
  ... # 事务中的其它语句

  BEGIN; # 此语句会隐式的提交前边语句所属于的事务
  ```

  或者当前的 autocommit 系统变量的值为 OFF ，我们手动把它调为 ON 时，也会隐式的提交前边语句所属的事务。

  或者使用 LOCK TABLES 、 UNLOCK TABLES 等关于锁定的语句也会隐式的提交前边语句所属的事务。

- 加载数据的语句

  比如我们使用 LOAD DATA 语句来批量往数据库中导入数据时，也会隐式的提交前边语句所属的事务。

- 关于 MySQL 复制的一些语句

  使用 START SLAVE 、 STOP SLAVE 、 RESET SLAVE 、 CHANGE MASTER TO 等语句时也会隐式的提交前边语句所属的事务。

- 其它的一些语句

  使用 ANALYZE TABLE 、 CACHE INDEX 、 CHECK TABLE 、 FLUSH 、 LOAD INDEX INTO CACHE 、 OPTIMIZE TABLE 、 REPAIR TABLE 、 RESET 等语句也会隐式的提交前边语句所属的事务。

  小贴士：
  上边提到的一些语句，如果你都认识并且知道是干嘛用的那再好不过了，不认识也不要气馁，这里写出来只是为了内容的完整性，把可能会导致事务隐式提交的情况都列举一下，具体每个语句都是干嘛用的等我们遇到了再说哈。

19.3.7 保存点

如果你开启了一个事务，并且已经敲了很多语句，忽然发现上一条语句有点问题，你只好使用 ROLLBACK 语句来让数据库状态恢复到事务执行之前的样子，然后一切从头再来，总有一种一夜回到解放前的感觉。所以设计数据库的大叔们提出了一个 保存点 （英文： savepoint ）的概念，就是在事务对应的数据库语句中打几个点，我们在调用 ROLLBACK 语句时可以指定会滚到哪个点，而不是回到最初的原点。定义保存点的语法如下：

```
SAVEPOINT 保存点名称;
```

当我们想回滚到某个保存点时，可以使用下边这个语句（下边语句中的单词 WORK 和 SAVEPOINT 是可有可无的）：

```
ROLLBACK [WORK] TO [SAVEPOINT] 保存点名称;
```

不过如果 ROLLBACK 语句后边不跟随保存点名称的话，会直接回滚到事务执行之前的状态。

如果我们想删除某个保存点，可以使用这个语句：

```
RELEASE SAVEPOINT 保存点名称;
```

下边还是以狗哥向猫爷转账10元的例子展示一下 保存点 的用法，在执行完扣除狗哥账户的钱 10 元的语句之后打一个 保存点 ：

```
mysql> SELECT * FROM account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |      11 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
2 rows in set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE account SET balance = balance - 10 WHERE id = 1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SAVEPOINT s1;    # 一个保存点
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |       1 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
2 rows in set (0.00 sec)

mysql> UPDATE account SET balance = balance + 1 WHERE id = 2; # 更新错了
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> ROLLBACK TO s1;  # 回滚到保存点s1处
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM account;
+----+--------+---------+
| id | name   | balance |
+----+--------+---------+
|  1 | 狗哥   |       1 |
|  2 | 猫爷   |       2 |
+----+--------+---------+
2 rows in set (0.00 sec)
```

20 第20章 说过的话就一定要办到-redo日志（上）

标签： MySQL是怎样运行的

20.1 事先说明

本文以及接下来的几篇文章将会频繁的使用到我们前边唠叨的 InnoDB 记录行格式、页面格式、索引原理、表空间的组成等各种基础知识，如果大家对这些东西理解的不透彻，那么阅读下边的文字可能会有些吃力，为保证您的阅读体验，请确保自己已经掌握了我前边唠叨的这些知识。

20.2 redo日志是个啥

我们知道 InnoDB 存储引擎是以页为单位来管理存储空间的，我们进行的增删改查操作其实本质上都是在访问页面（包括读页面、写页面、创建新页面等操作）。我们前边唠叨 Buffer Pool 的时候说过，在真正访问页面之前，需要把在磁盘上的页缓存到内存中的 Buffer Pool 之后才可以访问。但是在唠叨事务的时候又强调过一个称之为 持久性 的特性，就是说对于一个已经提交的事务，在事务提交后即使系统发生了崩溃，这个事务对数据库中所做的更改也不能丢失。但是如果我们只在内存的 Buffer Pool 中修改了页面，假设在事务提交后突然发生了某个故障，导致内存中的数据都失效了，那么这个已经提交了的事务对数据库中所做的更改也就跟着丢失了，这是我们所不能忍受的（想想ATM机已经提示狗哥转账成功，但之后由于服务器出现故障，重启之后猫爷发现自己没收到钱，猫爷就被砍死了）。那么如何保证这个 持久性 呢？一个很简单的做法就是在事务提交完成之前把该事务所修改的所有页面都刷新到磁盘，但是这个简单粗暴的做法有些问题：

- 刷新一个完整的数据页太浪费了

  有时候我们仅仅修改了某个页面中的一个字节，但是我们知道在 InnoDB 中是以页为单位来进行磁盘IO的，也就是说我们在该事务提交时不得不将一个完整的页面从内存中刷新到磁盘，我们又知道一个页面默认是16KB大小，只修改一个字节就要刷新16KB的数据到磁盘上显然是太浪费了。
- 随机IO刷起来比较慢

  一个事务可能包含很多语句，即使是一条语句也可能修改许多页面，倒霉催的是该事务修改的这些页面可能并不相邻，这就意味着在将某个事务修改的 Buffer Pool 中的页面刷新到磁盘时，需要进行很多的随机IO，随机IO比顺序IO要慢，尤其对于传统的机械硬盘来说。

咋办呢？再次回到我们的初心：我们只是想让已经提交了的事务对数据库中数据所做的修改永久生效，即使后来系统崩溃，在重启后也能把这种修改恢复出来。所以我们其实没有必要在每次事务提交时就把该事务在内存中修改过的全部页面刷新到磁盘，只需要把修改了哪些东西记录一下就好，比方说某个事务将系统表空间中的第100号页面中偏移量为1000处的那个字节的值 1 改成 2 我们只需要记录一下：

> 将第0号表空间的100号页面的偏移量为1000处的值更新为 2 。

这样我们在事务提交时，把上述内容刷新到磁盘中，即使之后系统崩溃了，重启之后只要按照上述内容所记录的步骤重新更新一下数据页，那么该事务对数据库中所做的修改又可以被恢复出来，也就意味着满足 持久性 的要求。因为在系统奔溃重启时需要按照上述内容所记录的步骤重新更新数据页，所以上述内容也被称之为 重做日志 ，英文名为 redo log ，我们也可以土洋结合，称之为 redo日志 。与在事务提交时将所有修改过的内存中的页面刷新到磁盘中相比，只将该事务执行过程中产生的 redo 日志刷新到磁盘的好处如下：

- redo 日志占用的空间非常小

  存储表空间ID、页号、偏移量以及需要更新的值所需的存储空间是很小的，关于 redo 日志的格式我们稍后会详细唠叨，现在只要知道一条 redo 日志占用的空间不是很大就好了。
- redo 日志是顺序写入磁盘的

  在执行事务的过程中，每执行一条语句，就可能产生若干条 redo 日志，这些日志是按照产生的顺序写入磁盘的，也就是使用顺序IO。

20.3 redo日志格式

通过上边的内容我们知道， redo 日志本质上只是记录了一下事务对数据库做了哪些修改。 设计 InnoDB 的大叔们针对事务对数据库的不同修改场景定义了多种类型的 redo 日志，但是绝大部分类型的 redo 日志都有下边这种通用的结构：



各个部分的详细释义如下：

- type ：该条 redo 日志的类型。

  在 MySQL 5.7.21 这个版本中，设计 InnoDB 的大叔一共为 redo 日志设计了53种不同的类型，稍后会详细介绍不同类型的 redo 日志。
- space ID ：表空间ID。
- page number ：页号。
- data ：该条 redo 日志的具体内容。

20.3.1 简单的redo日志类型

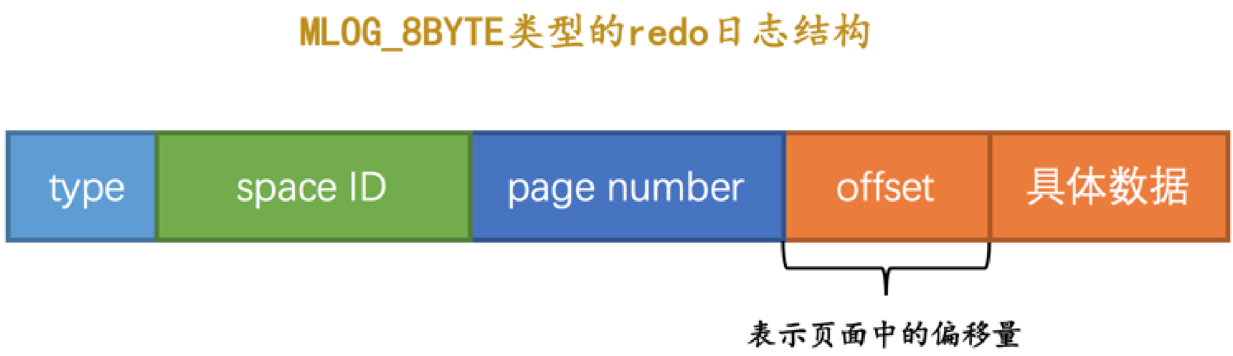
我们前边介绍 InnoDB 的记录行格式的时候说过，如果我们没有为某个表显式的定义主键，并且表中也没有定义 Unique 键，那么 InnoDB 会自动的为表添加一个称之为 row_id 的隐藏列作为主键。为这个 row_id 隐藏列赋值的方式如下：

- 服务器会在内存中维护一个全局变量，每当向某个包含隐藏的 row_id 列的表中插入一条记录时，就会把该变量的值当作新记录的 row_id 列的值，并且把该变量自增1。
- 每当这个变量的值为256的倍数时，就会将该变量的值刷新到系统表空间的页号为 7 的页面中一个称之为 Max Row ID 的属性处（我们前边介绍表空间结构时详细说过）。
- 当系统启动时，会将上边提到的 Max Row ID 属性加载到内存中，将该值加上256之后赋值给我们前边提到的全局变量（因为在上次关机时该全局变量的值可能大于 Max Row ID 属性值）。

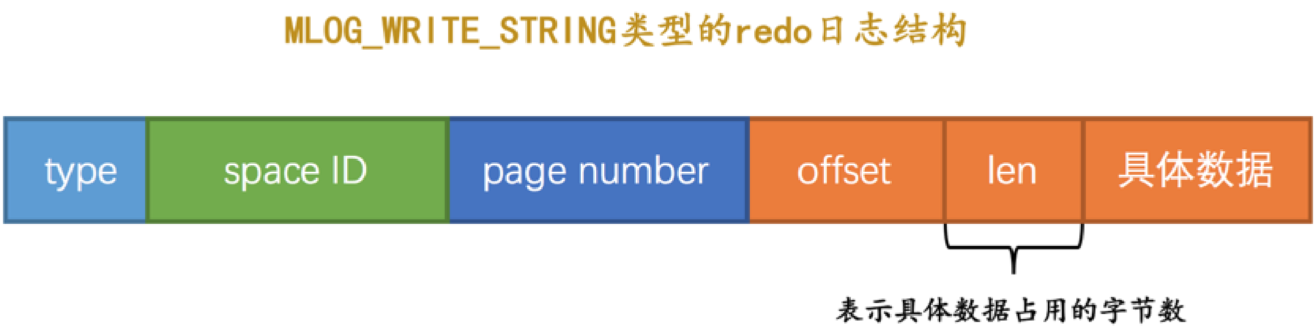
这个 Max Row ID 属性占用的存储空间是8个字节，当某个事务向某个包含 row_id 隐藏列的表插入一条记录，并且为该记录分配的 row_id 值为256的倍数时，就会向系统表空间页号为7的页面的相应偏移量处写入8个字节的值。但是我们要知道，这个写入实际上是在 Buffer Pool 中完成的，我们需要为这个页面的修改记录一条 redo 日志，以便在系统奔溃后能将已经提交的该事务对该页面所做的修改恢复出来。这种情况下对页面的修改是极其简单的， redo 日志中只需要记录一下在某个页面的某个偏移量处修改了几个字节的值，具体被修改的内容是啥就好了，设计 InnoDB 的大叔把这种极其简单的 redo 日志称之为 物理日志 ，并且根据在页面中写入数据的多少划分了几种不同的 redo 日志类型：

- MLOG_1BYTE （ type 字段对应的十进制数字为 1 ）：表示在页面的某个偏移量处写入1个字节的 redo 日志类型。
- MLOG_2BYTE （ type 字段对应的十进制数字为 2 ）：表示在页面的某个偏移量处写入2个字节的 redo 日志类型。
- MLOG_4BYTE （ type 字段对应的十进制数字为 4 ）：表示在页面的某个偏移量处写入4个字节的 redo 日志类型。
- MLOG_8BYTE （ type 字段对应的十进制数字为 8 ）：表示在页面的某个偏移量处写入8个字节的 redo 日志类型。
- MLOG_WRITE_STRING （ type 字段对应的十进制数字为 30 ）：表示在页面的某个偏移量处写入一串数据。

我们上边提到的 Max Row ID 属性实际占用8个字节的存储空间，所以在修改页面中的该属性时，会记录一条类型为 MLOG_8BYTE 的 redo 日志， MLOG_8BYTE 的 redo 日志结构如下所示：



其余 MLOG_1BYTE 、 MLOG_2BYTE 、 MLOG_4BYTE 类型的 redo 日志结构和 MLOG_8BYTE 的类似，只不过具体数据中包含对应个字节的数据罢了。 MLOG_WRITE_STRING 类型的 redo 日志表示写入一串数据，但是因为不能确定写入的具体数据占用多少字节，所以需要在日志结构中添加一个 len 字段：



> 小贴士：
> 只要将MLOG_WRITE_STRING类型的redo日志的len字段填充上1、2、4、8这些数字，就可以分别替代MLOG_1BYTE、MLOG_2BYTE、MLOG_4BYTE、MLOG_8BYTE这些类型的redo日志，为啥还要多此一举设计这么多类型呢？还不是因为省空间啊，能不写len字段就不写len字段，省一个字节算一个字节。

20.3.2 复杂一些的redo日志类型

有时候执行一条语句会修改非常多的页面，包括系统数据页面和用户数据页面（用户数据指的就是聚簇索引和二级索引对应的 B+ 树）。以一条 INSERT 语句为例，它除了要向 B+ 树的页面中插入数据，也可能更新系统数据 Max Row ID 的值，不过对于我们用户来说，平时更关心的是语句对 B+ 树所做更新：

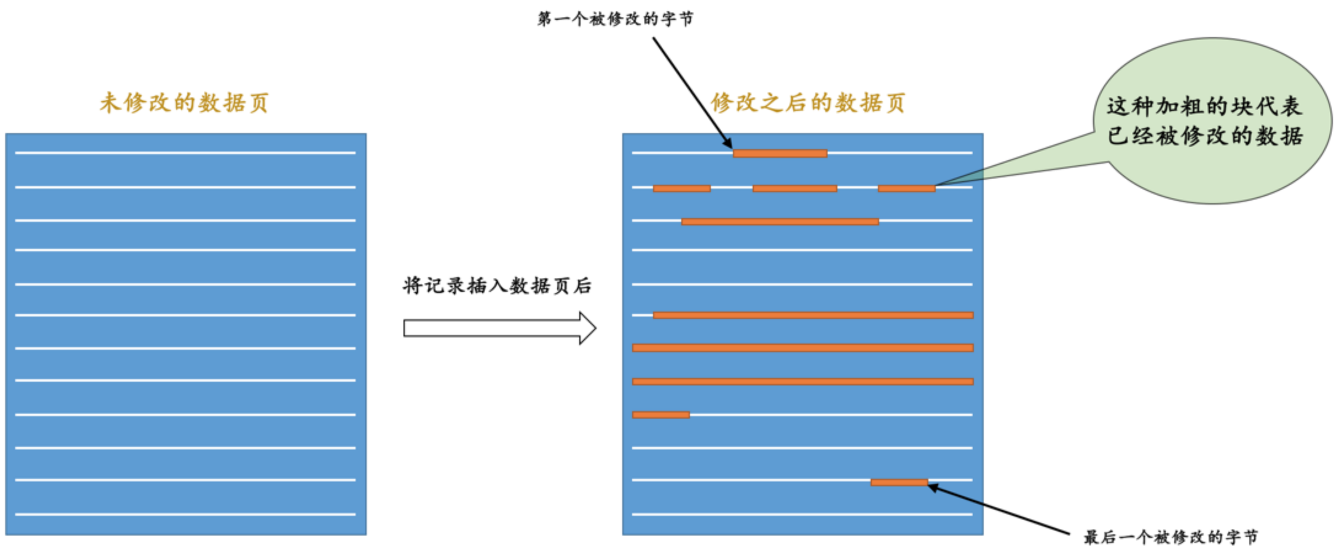
- 表中包含多少个索引，一条 INSERT 语句就可能更新多少棵 B+ 树。
- 针对某一棵 B+ 树来说，既可能更新叶子节点页面，也可能更新内节点页面，也可能创建新的页面（在该记录插入的叶子节点的剩余空间比较少，不足以存放该记录时，会进行页面的分裂，在内节点页面中添加 目录项记录 ）。

在语句执行过程中， INSERT 语句对所有页面的修改都得保存到 redo 日志中去。这句话说的比较轻巧，做起来可就比较麻烦了，比方说将记录插入到聚簇索引中时，如果定位到的叶子节点的剩余空间足够存储该记录时，那么只更新该叶子节点页面就好，那么只记录一条 MLOG_WRITE_STRING 类型的 redo 日志，表明在页面的某个偏移量

处增加了哪些数据就好了么？那就too young too naive了~ 别忘了一个数据页中除了存储实际的记录之后，还有什么 File Header 、 Page Header 、 Page Directory 等等部分（在唠叨数据页的章节有详细讲解），所以每往叶子节点代表的数据页里插入一条记录时，还有其他很多地方会跟着更新，比如说：

- 可能更新 Page Directory 中的槽信息。
- Page Header 中的各种页面统计信息，比如 PAGE_N_DIR_SLOTS 表示的槽数量可能会更改， PAGE_HEAP_TOP 代表的还未使用的空间最小地址可能会更改， PAGE_N_HEAP 代表的本页面中的记录数量可能会更改，吧啦吧啦，各种信息都可能会被修改。
- 我们知道在数据页里的记录是按照索引列从小到大的顺序组成一个单向链表的，每插入一条记录，还需要更新上一条记录的记录头信息中的 next_record 属性来维护这个单向链表。
- 还有别的吧啦吧啦的更新的地方，就不一一唠叨了...

画一个简易的示意图就像是这样：



说了这么多，就是想表达：把一条记录插入到一个页面时需要更改的地方非常多。这时我们如果使用上边介绍的简单的物理 redo 日志来记录这些修改时，可以有两种解决方案：

- 方案一：在每个修改的地方都记录一条 redo 日志。

  也就是如上图所示，有多少个加粗的块，就写多少条物理 redo 日志。这样子记录 redo 日志的缺点是显而易见的，因为被修改的地方是在太多了，可能记录的 redo 日志占用的空间都比整个页面占用的空间都多了~
- 方案二：将整个页面的 第一个被修改的字节 到 最后一个修改的字节 之间所有的数据当成是一条物理 redo 日志中的具体数据。

  从图中也可以看出来， 第一个被修改的字节 到 最后一个修改的字节 之间仍然有许多没有修改过的数据，我们把这些没有修改的数据也加入到 redo 日志中去岂不是太浪费了~

正因为上述两种使用物理 redo 日志的方式来记录某个页面中做了哪些修改比较浪费，设计 InnoDB 的大叔本着勤俭节约的初心，提出了一些新的 redo 日志类型，比如：

- MLOG_REC_INSERT （对应的十进制数字为 9 ）：表示插入一条使用非紧凑行格式的记录时的 redo 日志类型。
- MLOG_COMP_REC_INSERT （对应的十进制数字为 38 ）：表示插入一条使用紧凑行格式的记录时的 redo 日志类型。

  小贴士：
  Redundant是一种比较原始的行格式，它就是非紧凑的。而Compact、Dynamic以及Compressed行格式是较新的行格式，它们是紧凑的（占用更小的存储空间）。

- MLOG_COMP_PAGE_CREATE （ type 字段对应的十进制数字为 58 ）：表示创建一个存储紧凑行格式记录的页面的 redo 日志类型。
- MLOG_COMP_REC_DELETE （ type 字段对应的十进制数字为 42 ）：表示删除一条使用紧凑行格式记录的 redo 日志类型。
- MLOG_COMP_LIST_START_DELETE （ type 字段对应的十进制数字为 44 ）：表示从某条给定记录开始删除页面中的一系列使用紧凑行格式记录的 redo 日志类型。
- MLOG_COMP_LIST_END_DELETE （ type 字段对应的十进制数字为 43 ）：与 MLOG_COMP_LIST_START_DELETE 类型的 redo 日志呼应，表示删除一系列记录直到 MLOG_COMP_LIST_END_DELETE 类型的 redo 日志对应的记录为止。

  小贴士：
  我们前边唠叨InnoDB数据页格式的时候重点强调过，数据页中的记录是按照索引列大小的顺序组成单向链表的。有时候我们会有删除索引列的值在某个区间范围内的所有记录的需求，这时候如果我们每删除一条记录就写一条redo日志的话，效率可能有点低，所以提出MLOG_COMP_LIST_START_DELETE和MLOG_COMP_LIST_END_DELETE类型的redo日志，可以很大程度上减少redo日志的条数。

- MLOG_ZIP_PAGE_COMPRESS （ type 字段对应的十进制数字为 51 ）：表示压缩一个数据页的 redo 日志类型。
- ……还有很多很多种类型，这就不列举了，等用到再说哈～

这些类型的 redo 日志既包含 物理 层面的意思，也包含 逻辑 层面的意思，具体指：

- 物理层面看，这些日志都指明了对哪个表空间的哪个页进行了修改。
- 逻辑层面看，在系统奔溃重启时，并不能直接根据这些日志里的记载，将页面内的某个偏移量处恢复成某个数据，而是需要调用一些事先准备好的函数，执行完这些函数后才可以将页面恢复成系统奔溃前的样子。

大家看到这可能有些懵逼，我们还是以类型为 MLOG_COMP_REC_INSERT 这个代表插入一条使用紧凑行格式的记录时的 redo 日志为例来理解一下我们上边所说的 物理 层面和 逻辑 层面到底是个啥意思。废话少说，直接看一下这个类型为 MLOG_COMP_REC_INSERT 的 redo 日志的结构（由于字段太多了，我们把它们竖着看效果好些）：

MLOG_COMP_REC_INSERT 类型的redo日志结构



这个类型为 MLOG_COMP_REC_INSERT 的 redo 日志结构有几个地方需要大家注意：

- 我们前边在唠叨索引的时候说过，在一个数据页里，不论是叶子节点还是非叶子节点，记录都是按照索引列从小到大的顺序排序的。对于二级索引来说，当索引列的值相同时，记录还需要按照主键值进行排序。图中 n_uniques 的值的含义是在一条记录中，需要几个字段的值才能确保记录的唯一性，这样当插入一条记录时就可以按照记录的前 n_uniques 个字段进行排序。对于聚簇索引来说， n_uniques 的值为主键的列数，对于其他二级索引来说，该值为索引列数+主键列数。这里需要注意的是，唯一二级索引的值可能为 NULL ，所以该值仍然为索引列数+主键列数。
- field1_len ~ fieldn_len 代表着该记录若干个字段占用存储空间的大小，需要注意的是，这里不管该字段的类型是固定长度大小的（比如 INT ），还是可变长度大小（比如 VARCHAR(M) ）的，该字段占用的大小始终要写入 redo 日志中。
- offset 代表的是该记录的前一条记录在页面中的地址。为啥要记录前一条记录的地址呢？这是因为每向数据页插入一条记录，都需要修改该页面中维护的记录链表，每条记录的 记录头信息 中都包含一个称为 next_record 的属性，所以在插入新记录时，需要修改前一条记录的 next_record 属性。
- 我们知道一条记录其实由 额外信息 和 真实数据 这两部分组成，这两个部分的总大小就是一条记录占用存储空间的总大小。通过 end_seg_len 的值可以间接的计算出一条记录占用存储空间的总大小，为啥不直接存储一条记录占用存储空间的总大小呢？这是因为写 redo 日志是一个非常频繁的操作，设计 InnoDB 的大叔想方设法想减小 redo 日志本身占用的存储空间大小，所以想了一些弯弯绕的算法来实现这个目标，

end_seg_len 这个字段就是为了节省 redo 日志存储空间而提出来的。至于具体设计 InnoDB 的大叔到底是用了什么神奇魔法减小 redo 日志大小的，我们这就不多唠叨了，因为的确有那么一丢丢小复杂，说清楚还是有一点点麻烦的，而且说明白了也没啥用。
- mismatch_index 的值也是为了节省 redo 日志的大小而设立的，大家可以忽略。

很显然这个类型为 MLOG_COMP_REC_INSERT 的 redo 日志并没有记录 PAGE_N_DIR_SLOTS 的值修改为了啥，PAGE_HEAP_TOP 的值修改为了啥， PAGE_N_HEAP 的值修改为了啥等等这些信息，而只是把在本页面中插入一条记录所有必备的要素记了下来，之后系统奔溃重启时，服务器会调用相关向某个页面插入一条记录的那个函数，而 redo 日志中的那些数据就可以被当成是调用这个函数所需的参数，在调用完该函数后，页面中的 PAGE_N_DIR_SLOTS 、 PAGE_HEAP_TOP 、 PAGE_N_HEAP 等等的值也就都被恢复到系统奔溃前的样子了。这就是所谓的 逻辑 日志的意思。

20.3.3 redo日志格式小结

虽然上边说了一大堆关于 redo 日志格式的内容，但是如果你不是为了写一个解析 redo 日志的工具或者自己开发一套 redo 日志系统的话，那就没必要把 InnoDB 中的各种类型的 redo 日志格式都研究的透透的，没那个必要。上边我只是象征性的介绍了几种类型的 redo 日志格式，目的还是想让大家明白：redo日志会把事务在执行过程中对数据库所做的所有修改都记录下来，在之后系统奔溃重启后可以把事务所做的任何修改都恢复出来。

> 小贴士：
> 为了节省redo日志占用的存储空间大小，设计InnoDB的大叔对redo日志中的某些数据还可能进行压缩处理，比方说spacd ID和page number一般占用4个字节来存储，但是经过压缩后，可能使用更小的空间来存储。具体压缩算法就不唠叨了。

20.4 Mini-Transaction

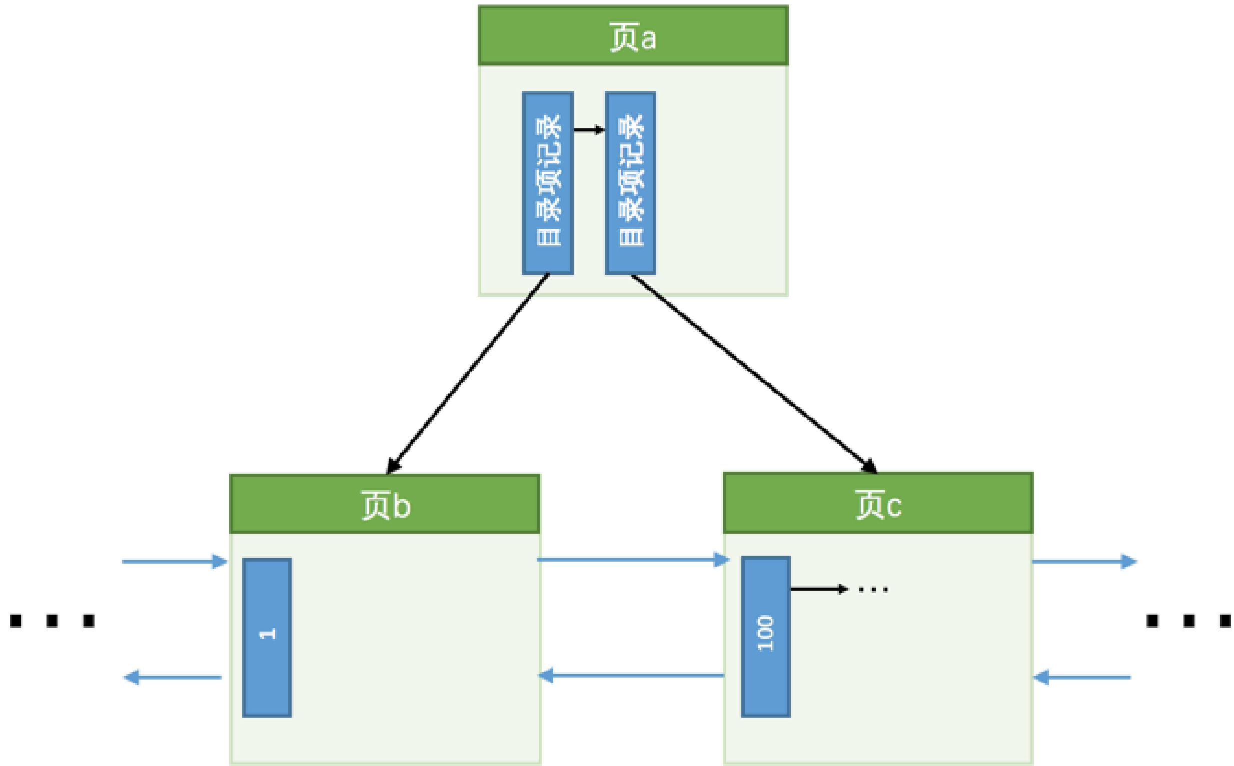
20.4.1 以组的形式写入redo日志

语句在执行过程中可能修改若干个页面。比如我们前边说的一条 INSERT 语句可能修改系统表空间页号为 7 的页面的 Max Row ID 属性（当然也可能更新别的系统页面，只不过我们没有都列举出来而已），还会更新聚簇索引和二级索引对应 B+ 树中的页面。由于对这些页面的更改都发生在 Buffer Pool 中，所以在修改完页面之后，需要记录一下相应的 redo 日志。在执行语句的过程中产生的 redo 日志被设计 InnoDB 的大叔人为的划分成了若干个不可分割的组，比如：

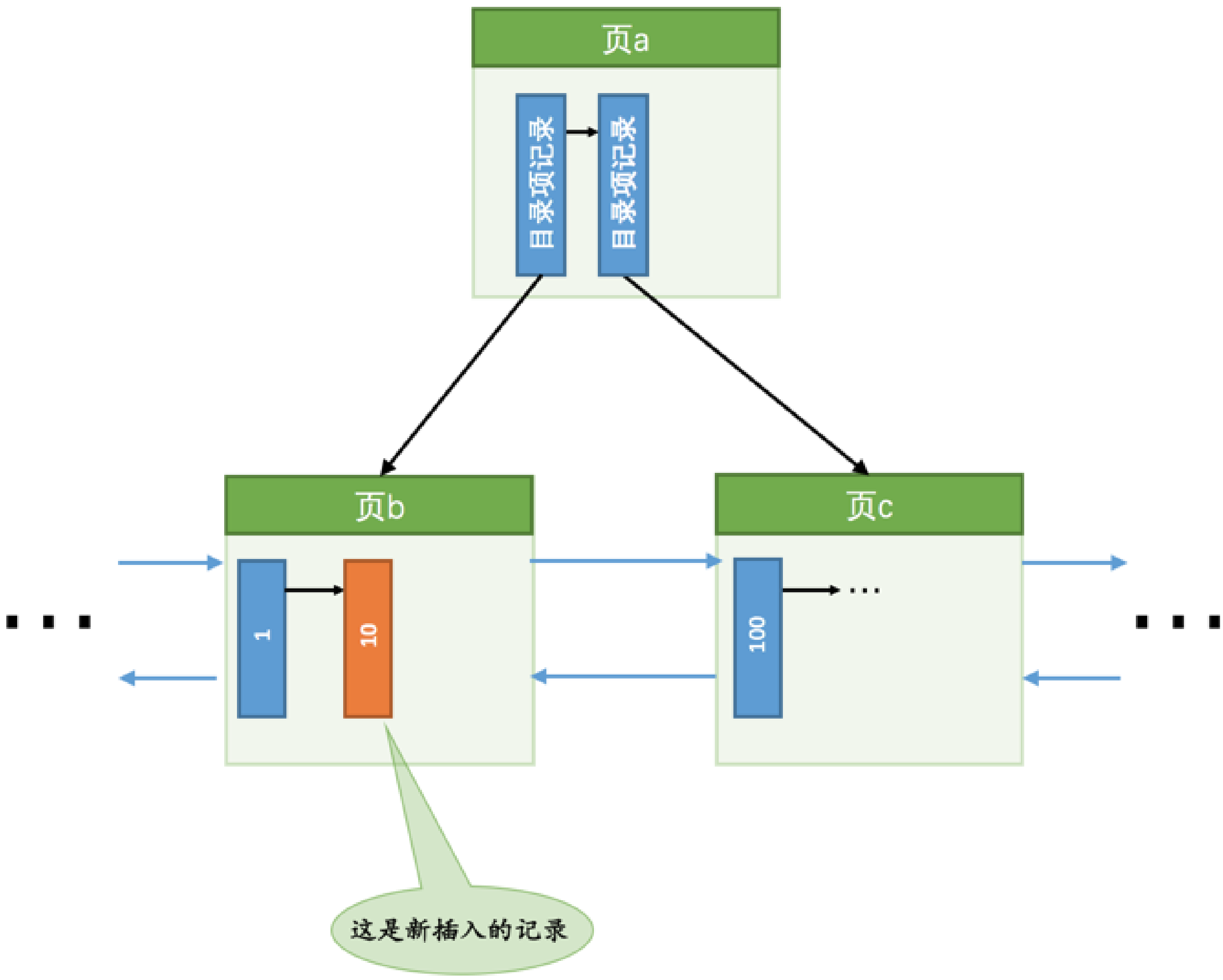
- 更新 Max Row ID 属性时产生的 redo 日志是不可分割的。
- 向聚簇索引对应 B+ 树的页面中插入一条记录时产生的 redo 日志是不可分割的。
- 向某个二级索引对应 B+ 树的页面中插入一条记录时产生的 redo 日志是不可分割的。
- 还有其他的一些对页面的访问操作时产生的 redo 日志是不可分割的。。。

怎么理解这个 不可分割 的意思呢？我们以向某个索引对应的 B+ 树插入一条记录为例，在向 B+ 树中插入这条记录之前，需要先定位到这条记录应该被插入到哪个叶子节点代表的数据页中，定位到具体的数据页之后，有两种可能的情况：

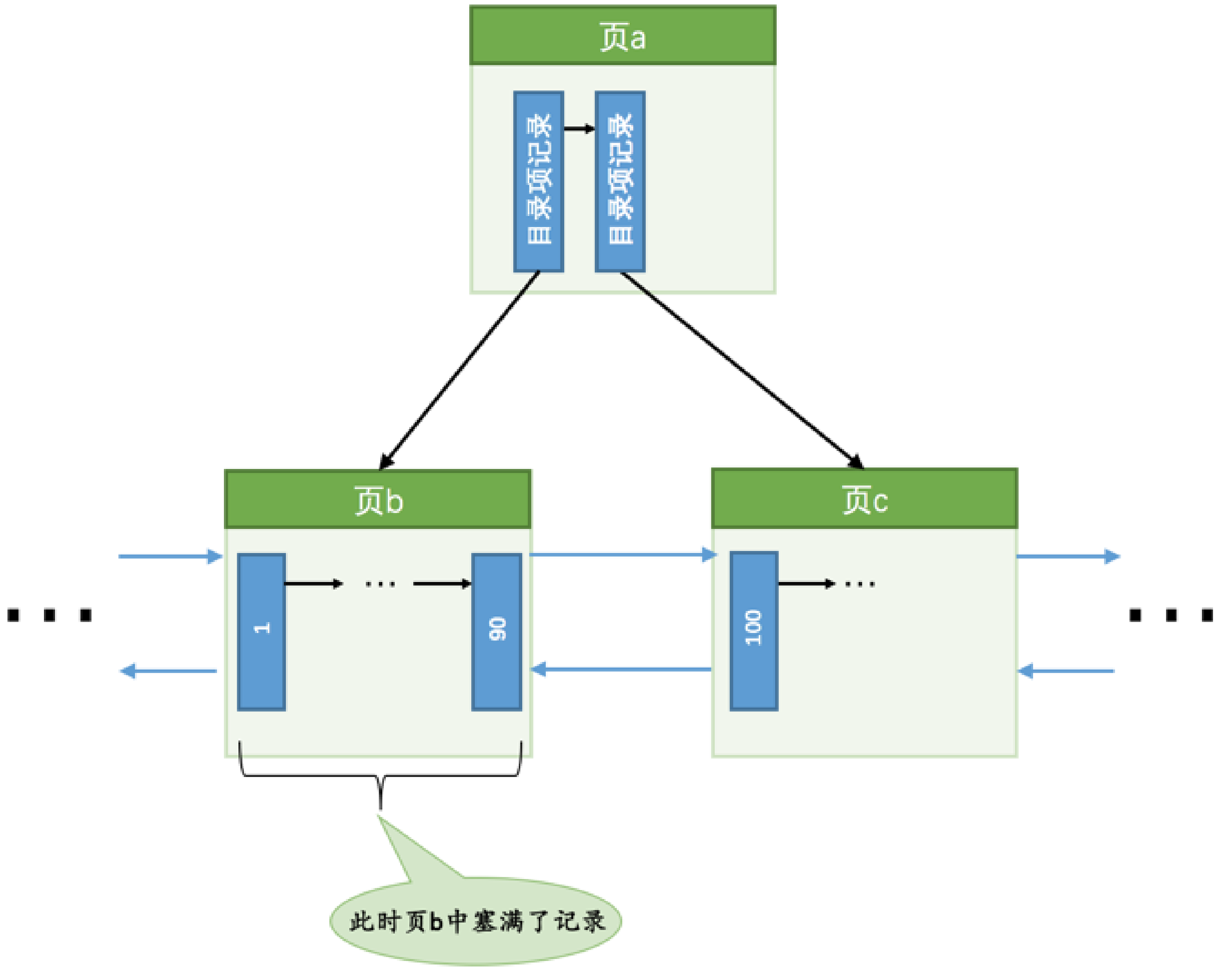
- 情况一：该数据页的剩余的空闲空间充足，足够容纳这一条待插入记录，那么事情很简单，直接把记录插入到这个数据页中，记录一条类型为 MLOG_COMP_REC_INSERT 的 redo 日志就好了，我们把这种情况称之为 乐观插入 。假如某个索引对应的 B+ 树长这样：



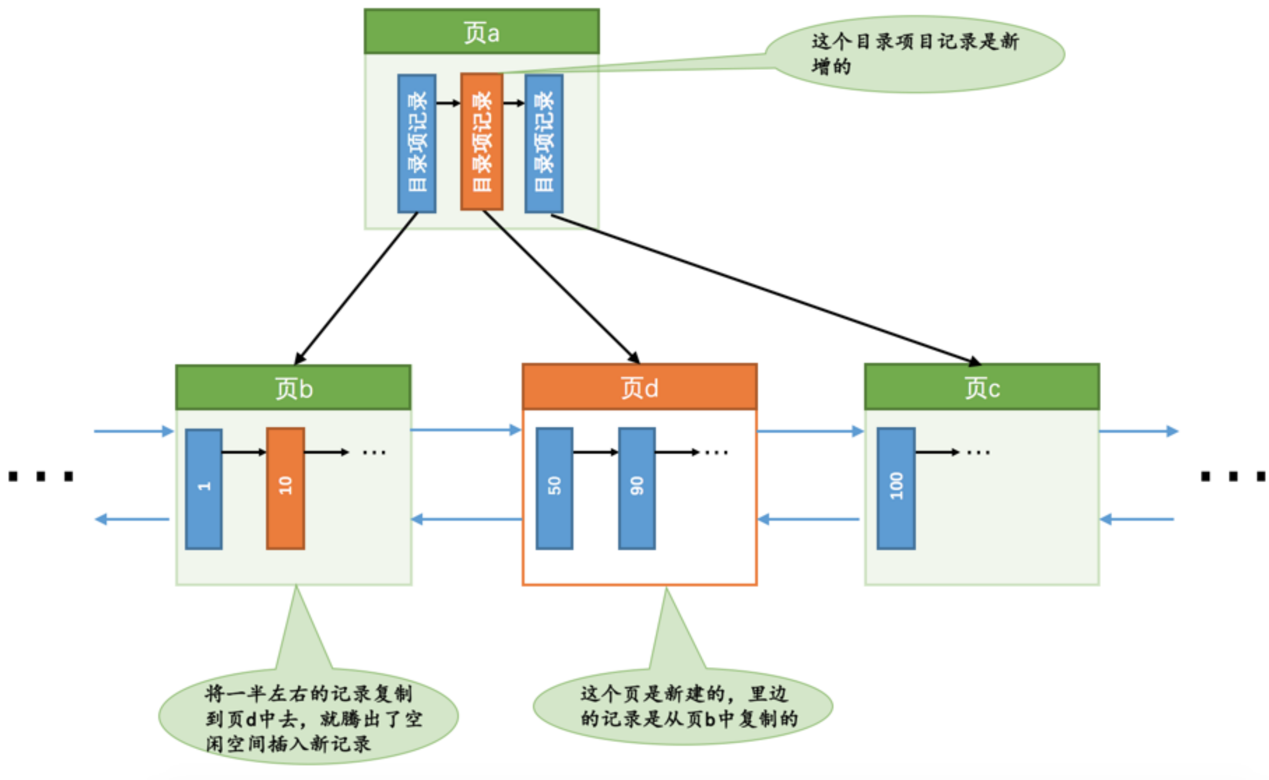
现在我们要插入一条键值为 10 的记录，很显然需要被插入到 页b 中，由于 页b 现在有足够的空间容纳一条记录，所以直接将该记录插入到 页b 中就好了，就像这样：



- 情况二：该数据页剩余的空闲空间不足，那么事情就悲剧了，我们前边说过，遇到这种情况要进行所谓的 页分裂 操作，也就是新建一个叶子节点，然后把原先数据页中的一部分记录复制到这个新的数据页中，然后再把记录插入进去，把这个叶子节点插入到叶子节点链表中，最后还要在内节点中添加一条 目录项记录 指向这个新创建的页面。很显然，这个过程要对多个页面进行修改，也就意味着会产生多条 redo 日志，我们把这种情况称之为 悲观插入 。假如某个索引对应的 B+ 树长这样：



现在我们要插入一条键值为 10 的记录，很显然需要被插入到 页b 中，但是从图中也可以看出来，此时 页b 已经塞满了记录，没有更多的空闲空间来容纳这条新记录了，所以我们需要进行页面的分裂操作，就像这样：



如果作为内节点的 页a 的剩余空闲空间也不足以容纳增加一条 目录项记录 ，那需要继续做内节点 页a 的分裂操作，也就意味着会修改更多的页面，从而产生更多的 redo 日志。另外，对于 悲观插入 来说，由于需要新申请数据页，还需要改动一些系统页面，比方说要修改各种段、区的统计信息信息，各种链表的统计信息（比如什么 FREE 链表、 FSP_FREE_FRAG 链表吧啦吧啦我们在唠叨表空间那一章中介绍过的各种东东）等等等等，反正总共需要记录的 redo 日志有二、三十条。

> 小贴士：
> 其实不光是悲观插入一条记录会生成许多条redo日志，设计InnoDB的大叔为了其他的一些功能，在乐观插入时也可能产生多条redo日志（具体是为了什么功能我们就不多说了，要不篇幅就受不了了～）。

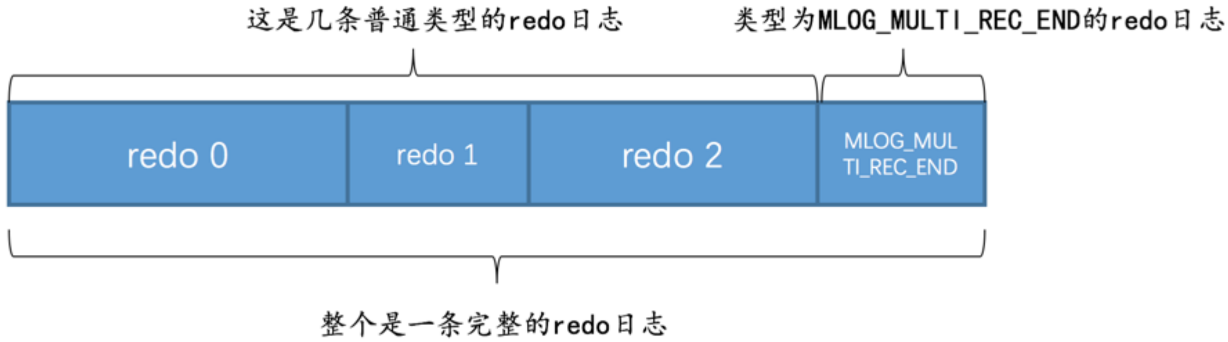
设计 InnoDB 的大叔们认为向某个索引对应的 B+ 树中插入一条记录的这个过程必须是原子的，不能说插了一半之后就停止了。比方说在悲观插入过程中，新的页面已经分配好了，数据也复制过去了，新的记录也插入到页面中了，可是没有向内节点中插入一条 目录项记录 ，这个插入过程就是不完整的，这样会形成一棵不正确的 B+ 树。我们知道 redo 日志是为了在系统奔溃重启时恢复崩溃前的状态，如果在悲观插入的过程中只记录了一部分 redo 日志，那么在系统奔溃重启时会将索引对应的 B+ 树恢复成一种不正确的状态，这是设计 InnoDB 的大叔们所不能忍受的。所以他们规定在执行这些需要保证原子性的操作时必须以 组 的形式来记录的 redo 日志，在进行系统奔溃重启恢复时，针对某个组中的 redo 日志，要么把全部的日志都恢复掉，要么一条也不恢复。怎么做到的呢？这得分情况讨论：

- 有的需要保证原子性的操作会生成多条 redo 日志，比如向某个索引对应的 B+ 树中进行一次悲观插入就需要生成许多条 redo 日志。

  如何把这些 redo 日志划分到一个组里边儿呢？设计 InnoDB 的大叔做了一个很简单的小把戏，就是在该组中的最后一条 redo 日志后边加上一条特殊类型的 redo 日志，该类型名称为 MLOG_MULTI_REC_END ， type 字段对应的十进制数字为 31 ，该类型的 redo 日志结构很简单，只有一个 type 字段：



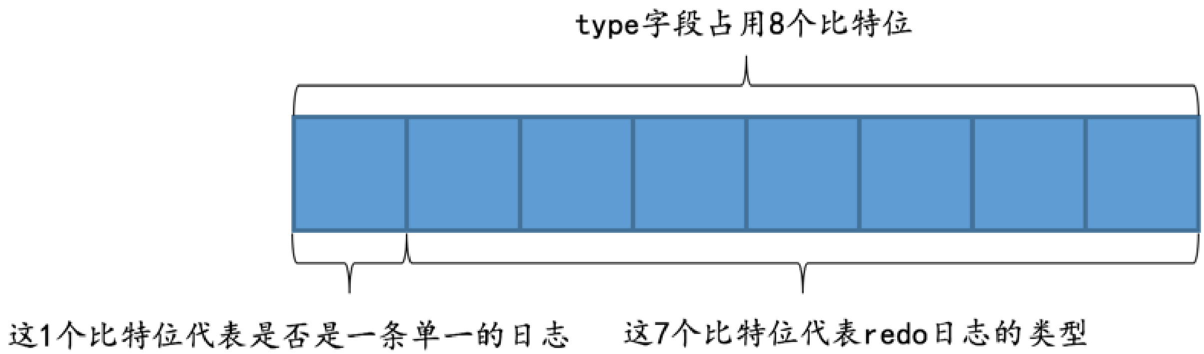
所以某个需要保证原子性的操作产生的一系列 redo 日志必须要以一个类型为 MLOG_MULTI_REC_END 结尾，就像这样：



这样在系统奔溃重启进行恢复时，只有当解析到类型为 MLOG_MULTI_REC_END 的 redo 日志，才认为解析到了一组完整的 redo 日志，才会进行恢复。否则的话直接放弃前边解析到的 redo 日志。

- 有的需要保证原子性的操作只生成一条 redo 日志，比如更新 Max Row ID 属性的操作就只会生成一条 redo 日志。

  其实在一条日志后边跟一个类型为 MLOG_MULTI_REC_END 的 redo 日志也是可以的，不过设计 InnoDB 的大叔比较勤俭节约，他们不想浪费一个比特位。别忘了虽然 redo 日志的类型比较多，但撑死了也就是几十种，是小于 127 这个数字的，也就是说我们用7个比特位就足以包括所有的 redo 日志类型，而 type 字段其实是占用1个字节的，也就是说我们可以省出来一个比特位用来表示该需要保证原子性的操作只产生单一的一条 redo 日志，示意图如下：

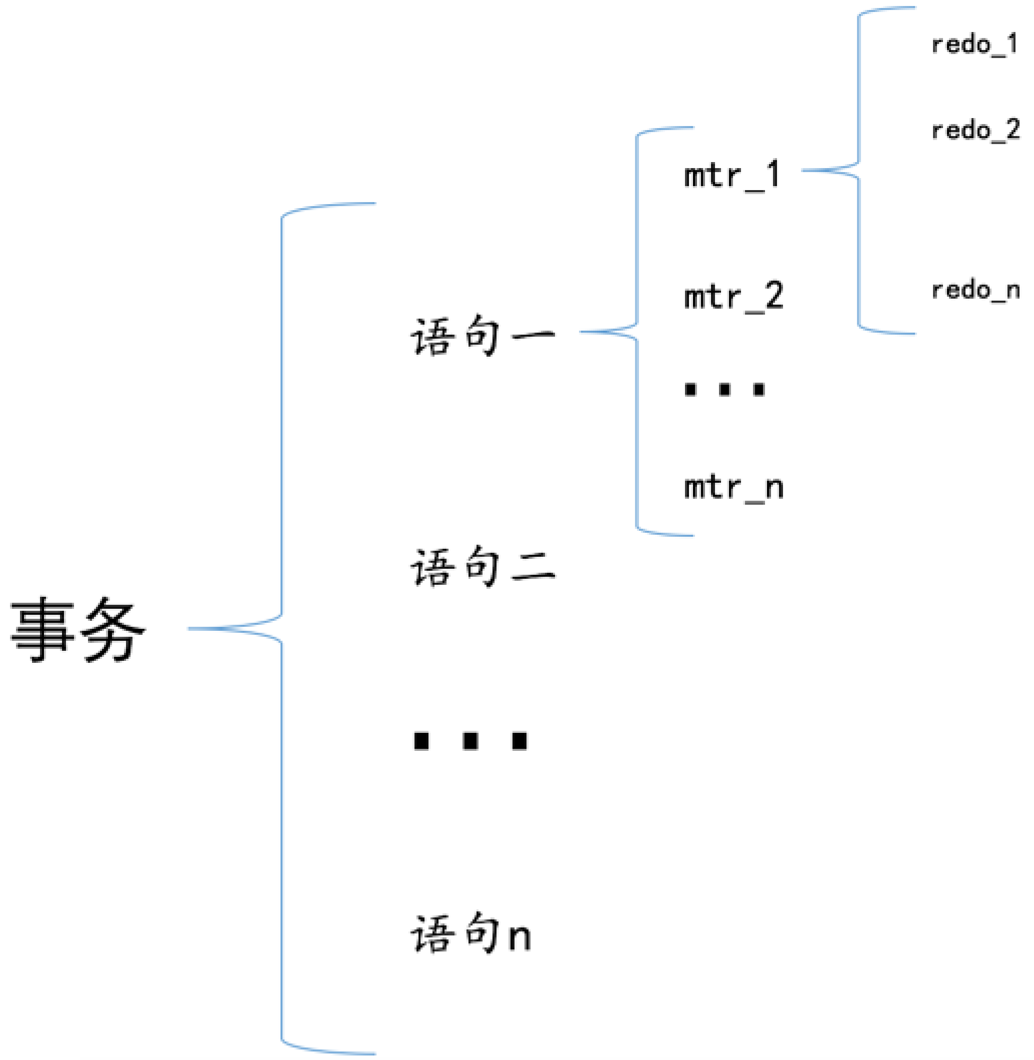


如果 type 字段的第一个比特位为 1 ，代表该需要保证原子性的操作只产生了单一的一条 redo 日志，否则表示该需要保证原子性的操作产生了一系列的 redo 日志。

20.4.2 Mini-Transaction的概念

设计 MySQL 的大叔把对底层页面中的一次原子访问的过程称之为一个 Mini-Transaction ，简称 mtr ，比如上边所说的修改一次 Max Row ID 的值算是一个 Mini-Transaction ，向某个索引对应的 B+ 树中插入一条记录的过程也算是一个 Mini-Transaction 。通过上边的叙述我们也知道，一个所谓的 mtr 可以包含一组 redo 日志，在进行奔溃恢复时这一组 redo 日志作为一个不可分割的整体。

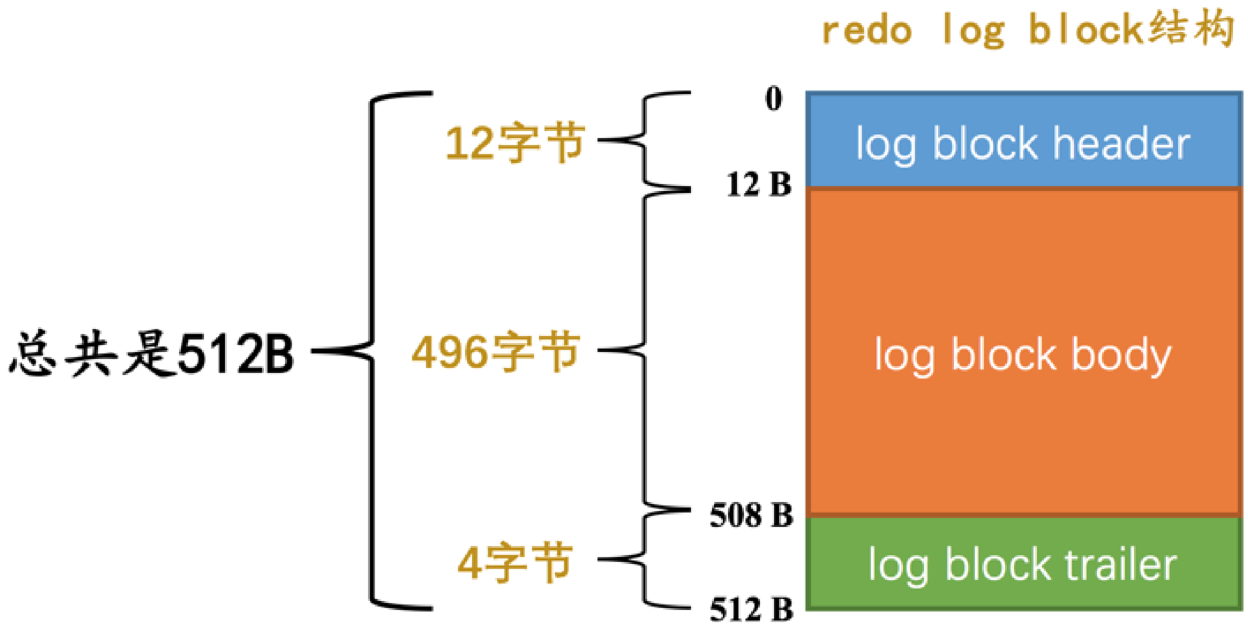
一个事务可以包含若干条语句，每一条语句其实是由若干个 mtr 组成，每一个 mtr 又可以包含若干条 redo 日志，画个图表示它们的关系就是这样：



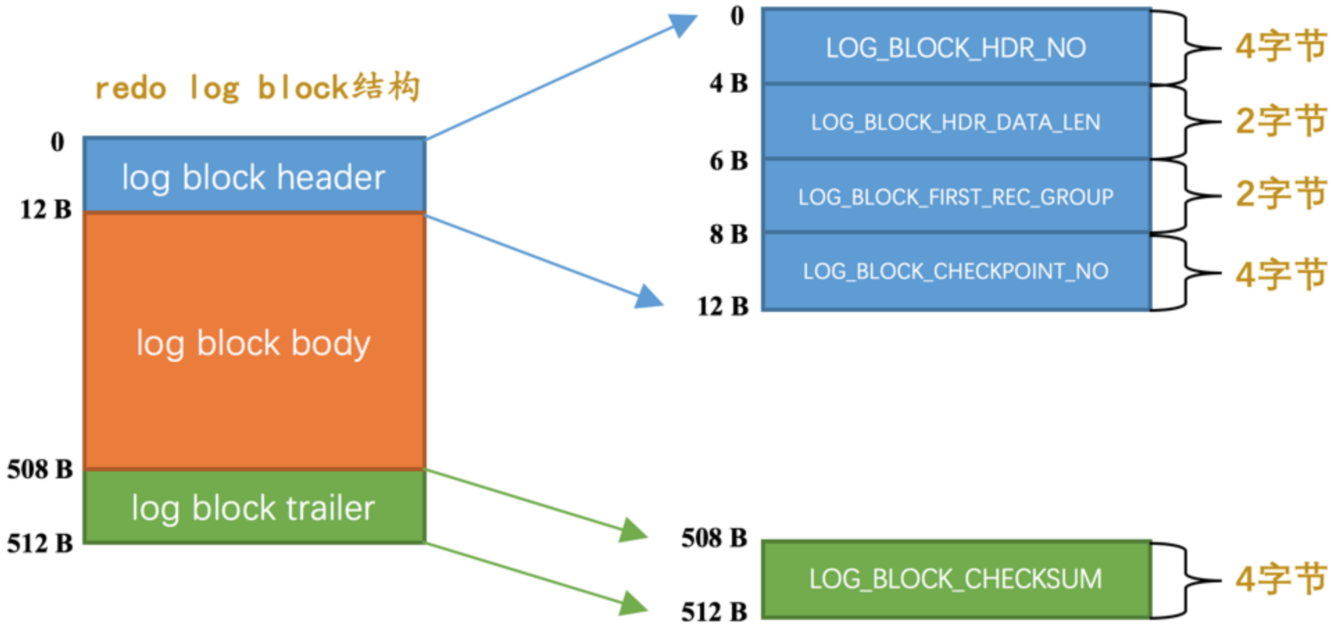
20.5 redo日志的写入过程

20.5.1 redo log block

设计 InnoDB 的大叔为了更好的进行系统奔溃恢复，他们把通过 mtr 生成的 redo 日志都放在了大小为 512字节的 页 中。为了和我们前边提到的表空间中的页做区别，我们这里把用来存储 redo 日志的页称为 block （你心里清楚页和block的意思其实差不多就行了）。一个 redo log block 的示意图如下：



真正的 redo 日志都是存储到占用 496 字节大小的 log block body 中，图中的 log block header 和 log block trailer 存储的是一些管理信息。我们来看看这些所谓的 管理信息 都是啥：



其中 log block header 的几个属性的意思分别如下：

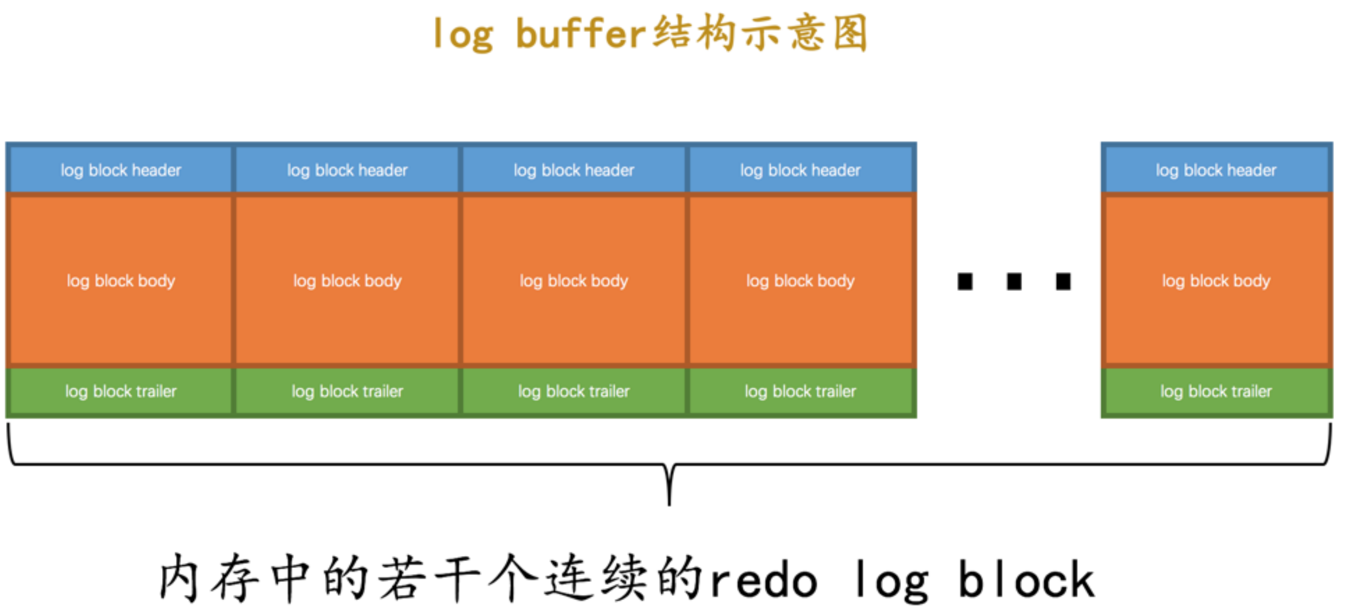
- LOG_BLOCK_HDR_NO ：每一个block都有一个大于0的唯一标号，本属性就表示该标号值。
- LOG_BLOCK_HDR_DATA_LEN ：表示block中已经使用了多少字节，初始值为 12 （因为 log block body 从第12个字节处开始）。随着往block中写入的redo日志越来也多，本属性值也跟着增长。如果 log block body 已经被全部写满，那么本属性的值被设置为 512 。
- LOG_BLOCK_FIRST_REC_GROUP ：一条 redo 日志也可以称之为一条 redo 日志记录（ redo log record ），一个 mtr 会生产多条 redo 日志记录，这些 redo 日志记录被称之为一个 redo 日志记录组（ redo log record group ）。 LOG_BLOCK_FIRST_REC_GROUP 就代表该block中第一个 mtr 生成的 redo 日志记录组的偏移量（其实也就是这个block里第一个 mtr 生成的第一条 redo 日志的偏移量）。
- LOG_BLOCK_CHECKPOINT_NO ：表示所谓的 checkpoint 的序号， checkpoint 是我们后续内容的重点，现在先不用清楚它的意思，稍安勿躁。

log block trailer 中属性的意思如下：

- LOG_BLOCK_CHECKSUM ：表示block的校验值，用于正确性校验，我们暂时不关心它。

20.5.2 redo日志缓冲区

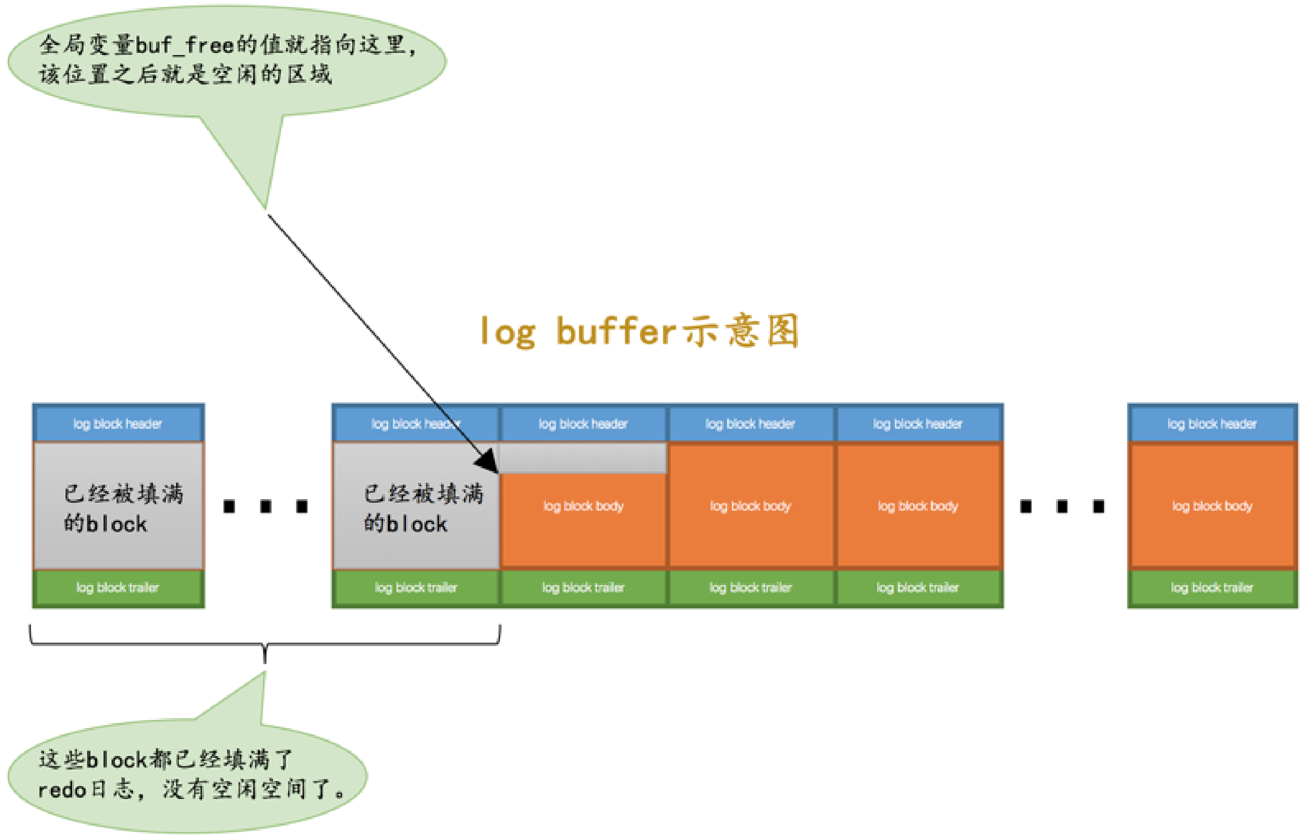
我们前边说过，设计 InnoDB 的大叔为了解决磁盘速度过慢的问题而引入了 Buffer Pool 。同理，写入 redo 日志时也不能直接直接写到磁盘上，实际上在服务器启动时就向操作系统申请了一大片称之为 redo log buffer 的连续内存空间，翻译成中文就是 redo日志缓冲区 ，我们也可以简称为 log buffer 。这片内存空间被划分成若干个连续的 redo log block ，就像这样：



我们可以通过启动参数 innodb_log_buffer_size 来指定 log buffer 的大小，在 MySQL 5.7.21 这个版本中，该启动参数的默认值为 16MB 。

20.5.3 redo日志写入log buffer

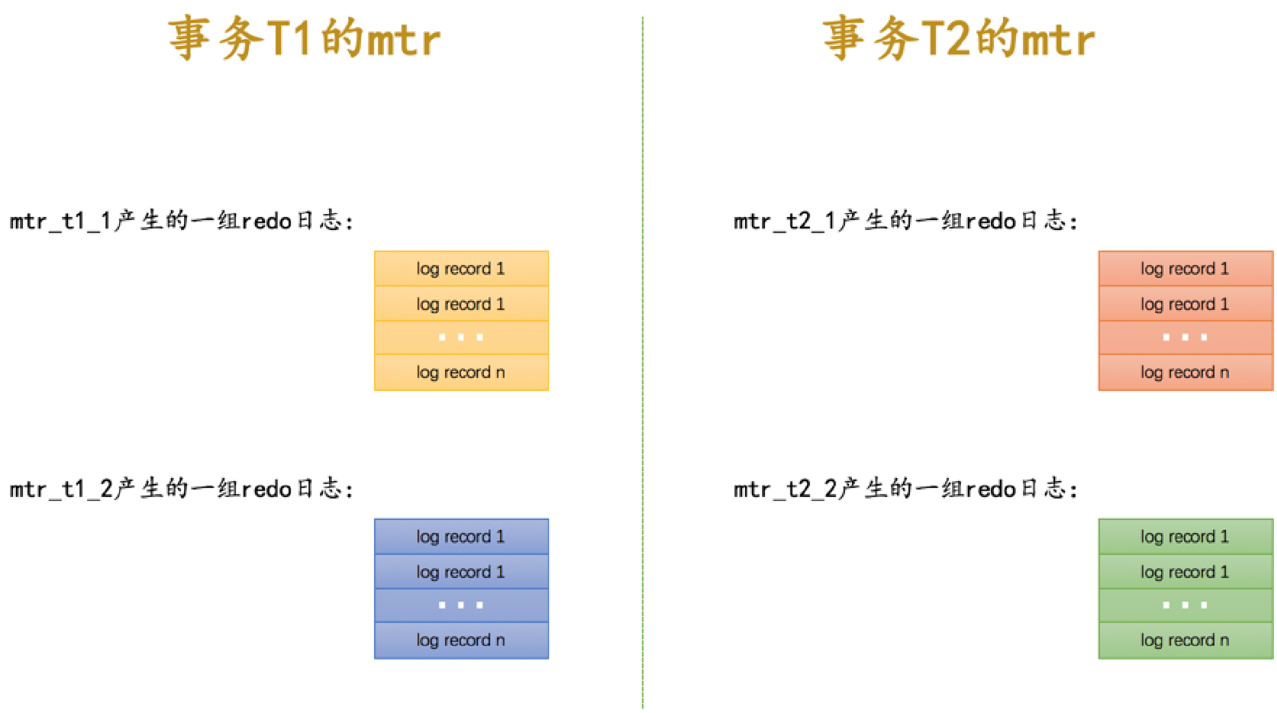
向 log buffer 中写入 redo 日志的过程是顺序的，也就是先往前边的block中写，当该block的空闲空间用完之后再往下一个block中写。当我们想往 log buffer 中写入 redo 日志时，第一个遇到的问题就是应该写在哪个 block 的哪个偏移量处，所以设计 InnoDB 的大叔特意提供了一个称之为 buf_free 的全局变量，该变量指明后续写入的 redo 日志应该写入到 log buffer 中的哪个位置，如图所示：



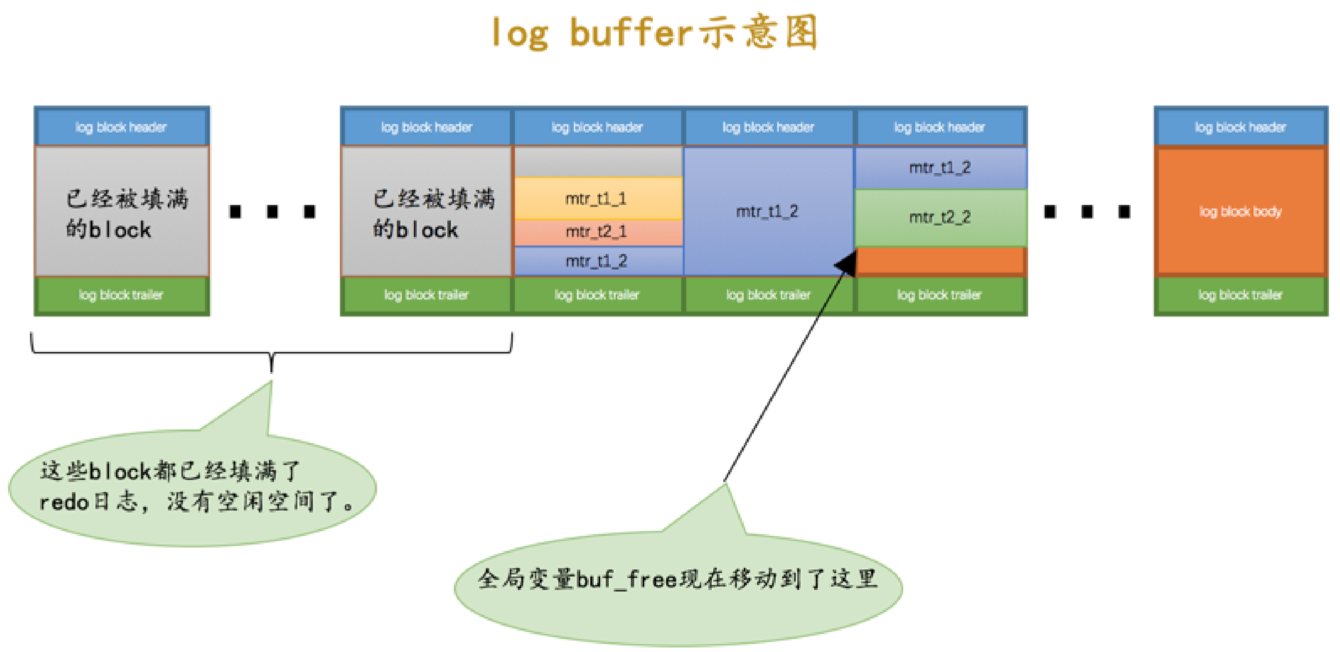
我们前边说过一个 mtr 执行过程中可能产生若干条 redo 日志，这些 redo 日志是一个不可分割的组，所以其实并不是每生成一条 redo 日志，就将其插入到 log buffer 中，而是每个 mtr 运行过程中产生的日志先暂时存到一个地方，当该 mtr 结束的时候，将过程中产生的一组 redo 日志再全部复制到 log buffer 中。我们现在假设有两个名为 T1 、 T2 的事务，每个事务都包含2个 mtr ，我们给这几个 mtr 命名一下：

- 事务 T1 的两个 mtr 分别称为 mtr_T1_1 和 mtr_T1_2 。
- 事务 T2 的两个 mtr 分别称为 mtr_T2_1 和 mtr_T2_2 。

每个 mtr 都会产生一组 redo 日志，用示意图来描述一下这些 mtr 产生的日志情况：



不同的事务可能是并发执行的，所以 T1 、 T2 之间的 mtr 可能是交替执行的。每当一个 mtr 执行完成时，伴随该 mtr 生成的一组 redo 日志就需要被复制到 log buffer 中，也就是说不同事务的 mtr 可能是交替写入 log buffer 的，我们画个示意图（为了美观，我们把一个 mtr 中产生的所有的 redo 日志当作一个整体来画）：



从示意图中我们可以看出来，不同的 mtr 产生的一组 redo 日志占用的存储空间可能不一样，有的 mtr 产生的 redo 日志量很少，比如 mtr_t1_1 、 mtr_t2_1 就被放到同一个block中存储，有的 mtr 产生的 redo 日志量非常大，比如 mtr_t1_2 产生的 redo 日志甚至占用了3个block来存储。

> 小贴士：
> 对照着上图，自己分析一下每个block的LOG_BLOCK_HDR_DATA_LEN、LOG_BLOCK_FIRST_REC_GROUP属性值都是什么哈～

21 第21章 说过的话就一定要办到-redo日志（下）

标签： MySQL是怎样运行的

21.1 redo日志文件

21.1.1 redo日志刷盘时机

我们前边说 mtr 运行过程中产生的一组 redo 日志在 mtr 结束时会被复制到 log buffer 中，可是这些日志总在内存里呆着也不是个办法，在一些情况下它们会被刷新到磁盘里，比如：

- log buffer 空间不足时

  log buffer 的大小是有限的（通过系统变量 innodb_log_buffer_size 指定），如果不停的往这个有限大小的 log buffer 里塞入日志，很快它就会被填满。设计 InnoDB 的大叔认为如果当前写入 log buffer 的 redo 日志量已经占满了 log buffer 总容量的大约一半左右，就需要把这些日志刷新到磁盘上。
- 事务提交时

我们前边说过之所以使用 redo 日志主要是因为它占用的空间少，还是顺序写，在事务提交时可以不把修改过的 Buffer Pool 页面刷新到磁盘，但是为了保证持久性，必须要把修改这些页面对应的 redo 日志刷新到磁盘。

- 后台线程不停的刷刷刷

  后台有一个线程，大约每秒都会刷新一次 log buffer 中的 redo 日志到磁盘。

- 正常关闭服务器时
- 做所谓的 checkpoint 时（我们现在没介绍过 checkpoint 的概念，稍后会仔细唠叨，稍安勿躁）
- 其他的一些情况...

21.1.2 redo日志文件组

MySQL 的数据目录（使用 SHOW VARIABLES LIKE 'datadir' 查看）下默认有两个名为 ib_logfile0 和 ib_logfile1 的文件， log buffer 中的日志默认情况下就是刷新到这两个磁盘文件中。如果我们对默认的 redo 日志文件不满意，可以通过下边几个启动参数来调节：

- innodb_log_group_home_dir

  该参数指定了 redo 日志文件所在的目录，默认值就是当前的数据目录。
- innodb_log_file_size

  该参数指定了每个 redo 日志文件的大小，在 MySQL 5.7.21 这个版本中的默认值为 48MB ，
- innodb_log_files_in_group

  该参数指定 redo 日志文件的个数，默认值为2，最大值为100。

从上边的描述中可以看到，磁盘上的 redo 日志文件不只一个，而是以一个 日志文件组 的形式出现的。这些文件以 ib_logfile[数字] （ 数字 可以是 0 、 1 、 2 ...）的形式进行命名。在将 redo 日志写入 日志文件组 时，是从 ib_logfile0 开始写，如果 ib_logfile0 写满了，就接着 ib_logfile1 写，同理， ib_logfile1 写满了就去写 ib_logfile2 ，依此类推。如果写到最后一个文件该咋办？那就重新转到 ib_logfile0 继续写，所以整个过程如下图所示：



总共的 redo 日志文件大小其实就是： innodb_log_file_size × innodb_log_files_in_group 。

> 小贴士：如果采用循环使用的方式向redo日志文件组里写数据的话，那岂不是要追尾，也就是后写入的redo日志覆盖掉前边写的redo日志？当然可能了！所以设计InnoDB的大叔提出了checkpoint的概念，稍后我们重点唠叨～

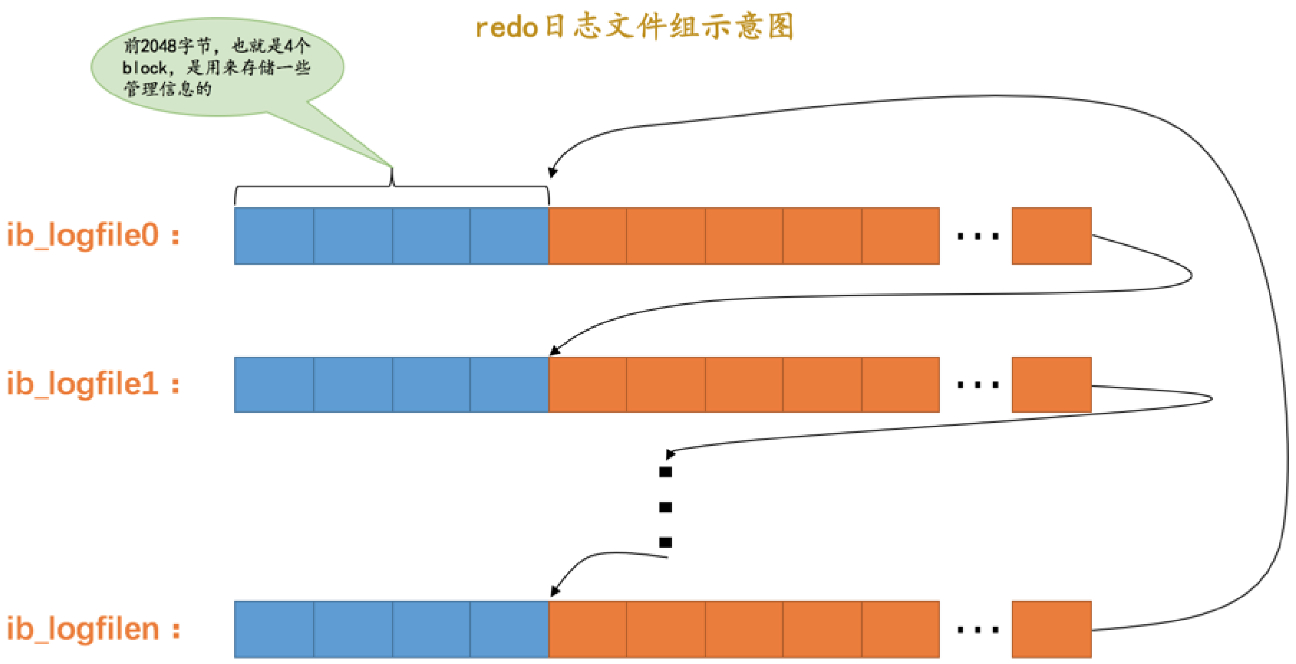
21.1.3 redo日志文件格式

我们前边说过 log buffer 本质上是一片连续的内存空间，被划分成了若干个 512 字节大小的 block 。将log buffer中的redo日志刷新到磁盘的本质就是把block的镜像写入日志文件中，所以 redo 日志文件其实也是由若干个 512 字节大小的block组成。

redo 日志文件组中的每个文件大小都一样，格式也一样，都是由两部分组成：

- 前2048个字节，也就是前4个block是用来存储一些管理信息的。
- 从第2048字节往后是用来存储 log buffer 中的block镜像的。

所以我们前边所说的 循环 使用redo日志文件，其实是从每个日志文件的第2048个字节开始算，画个示意图就是这样：

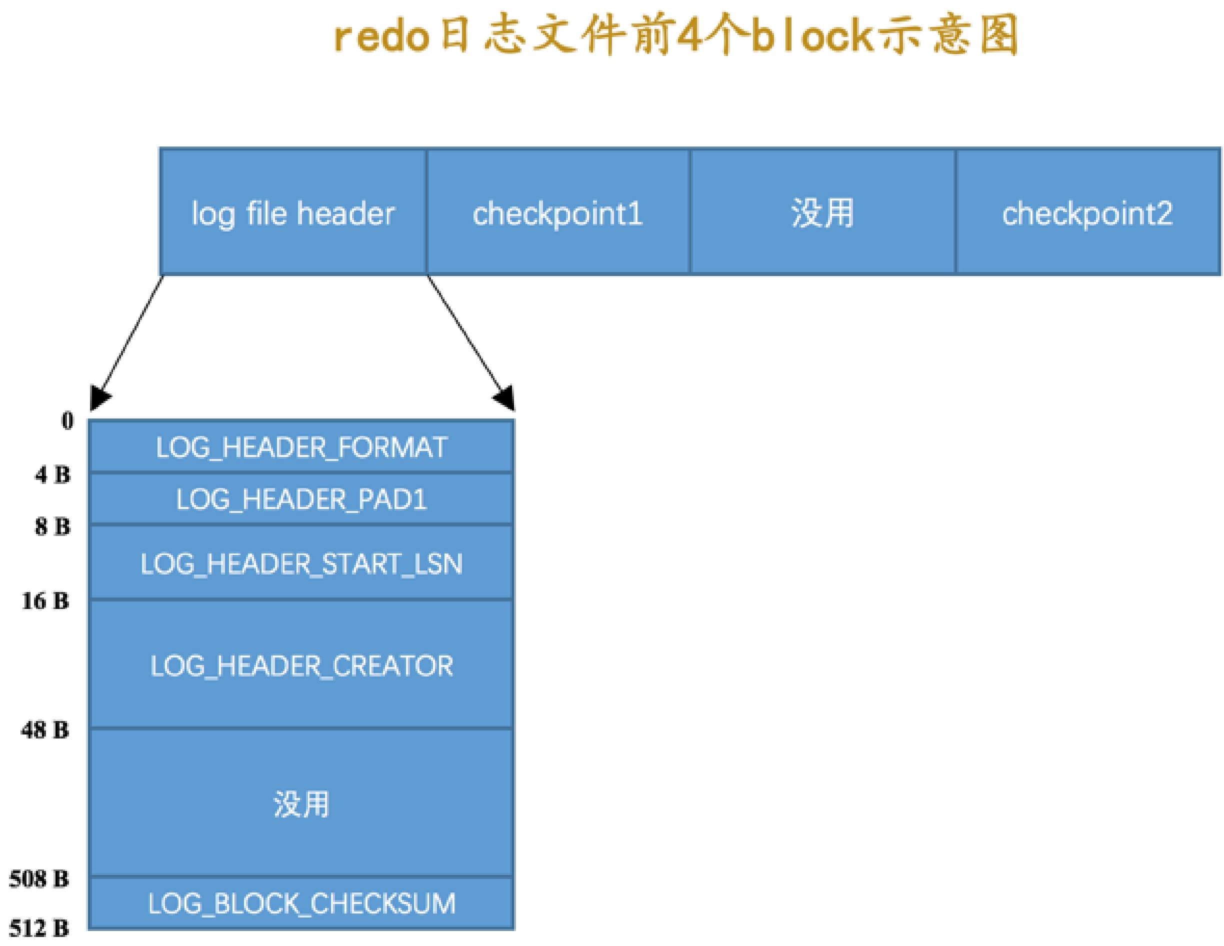


普通block的格式我们在唠叨 log buffer 的时候都说过了，就是 log block header 、 log block body 、 log block trialer 这三个部分，就不重复介绍了。这里需要介绍一下每个 redo 日志文件前2048个字节，也就是前4个特殊block的格式都是干嘛的，废话少说，先看图：



从图中可以看出来，这4个block分别是：

- log file header ：描述该 redo 日志文件的一些整体属性，看一下它的结构：

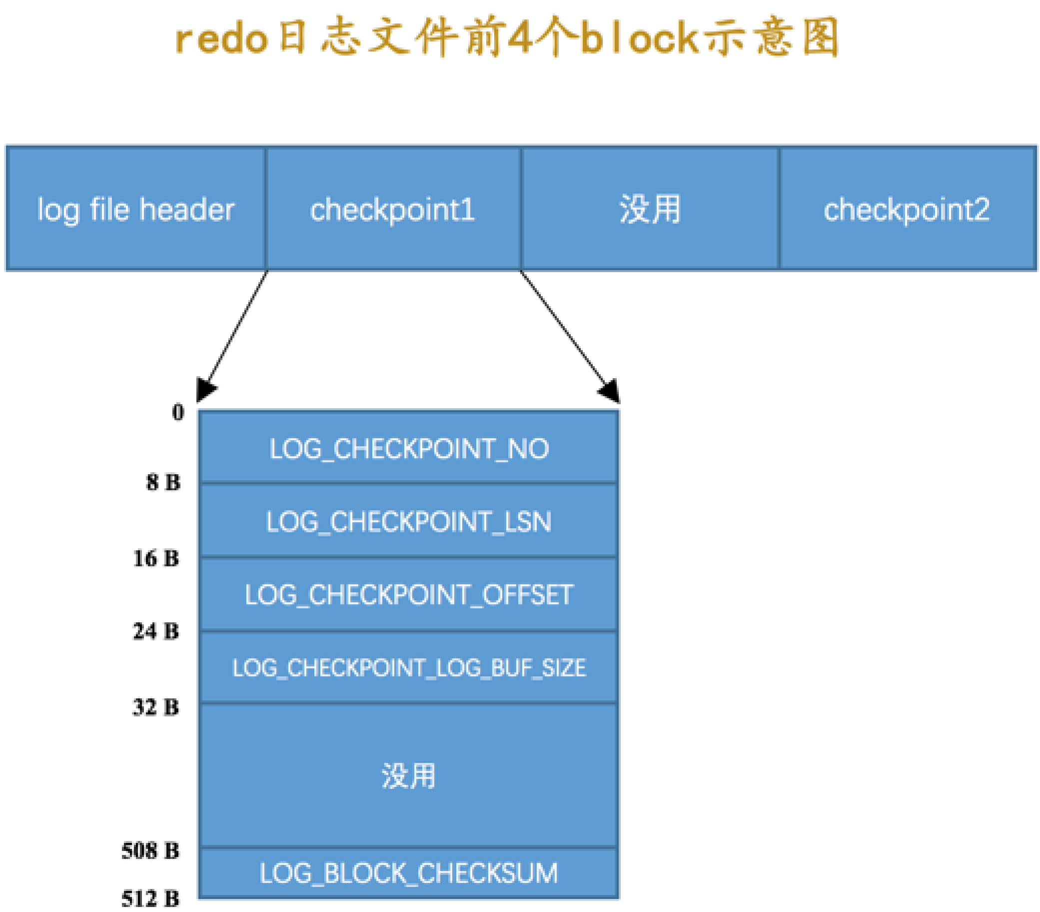


各个属性的具体释义如下：

|属性名|长度（单位：字节）|描述| |:--:|:--:|:--| | LOG_HEADER_FORMAT | 4 | redo 日志的版本，在 MySQL 5.7.21 中该值永远为1| | LOG_HEADER_PAD1 | 4 |做字节填充用的，没什么实际意义，忽略~| | LOG_HEADER_START_LSN | 8 |标记本 redo 日志文件开始的LSN值，也就是文件偏移量为2048字节初对应的LSN值（关于什么是LSN我们稍后再看哈，看不懂的先忽略）。| | LOG_HEADER_CREATOR | 32 |一个字符串，标记本 redo 日志文件的创建者是谁。正常运行时该值为 MySQL 的版本号，比如："MySQL 5.7.21"，使用 mysqlbackup 命令创建的 redo 日志文件的该值为 "ibbackup" 和创建时间。| | LOG_BLOCK_CHECKSUM | 4 |本block的校验值，所有block都有，我们不关心|

> 小贴士：
> 设计InnoDB的大叔对redo日志的block格式做了很多次修改，如果你阅读的其他书籍中发现上述的属性和你阅读书籍中的属性有些出入，不要慌，正常现象，忘记以前的版本吧。另外，LSN值我们后边才会介绍，现在千万别纠结LSN是个啥。

- checkpoint1 ：记录关于 checkpoint 的一些属性，看一下它的结构：



各个属性的具体释义如下：

|属性名|长度（单位：字节）|描述| |:--:|:--:|:--| | LOG_CHECKPOINT_NO | 8 |服务器做 checkpoint 的编号，每做一次 checkpoint ，该值就加1。| | LOG_CHECKPOINT_LSN | 8 |服务器做 checkpoint 结束时对应的 LSN 值，系统奔溃恢复时将从该值开始。| | LOG_CHECKPOINT_OFFSET | 8 |上个属性中的 LSN 值在 redo 日志文件组中的偏移量| | LOG_CHECKPOINT_LOG_BUF_SIZE | 8 |服务器在做 checkpoint 操作时对应的 log buffer 的大小| | LOG_BLOCK_CHECKSUM | 4 |本block的校验值，所有block都有，我们不关心|

> 小贴士：
> 现在看不懂上边这些关于checkpoint和LSN的属性的释义是很正常的，我就是想让大家对上边这些属性混个脸熟，后边我们后详细唠叨的。

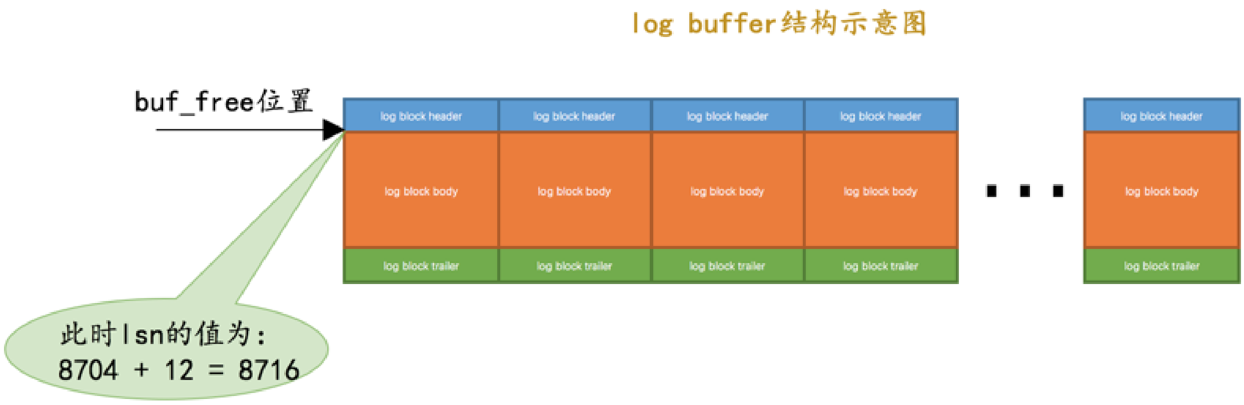
- 第三个block未使用，忽略～
- checkpoint2 ：结构和 checkpoint1 一样。

21.2 Log Sequeue Number

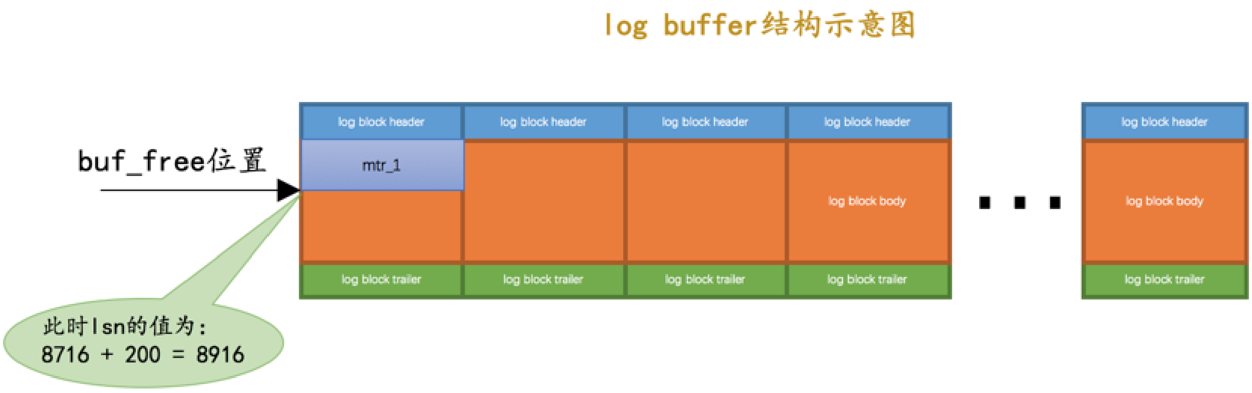
自系统开始运行，就不断的在修改页面，也就意味着会不断的生成 redo 日志。 redo 日志的量在不断的递增，就像人的年龄一样，自打出生起就不断递增，永远不可能缩减了。设计 InnoDB 的大叔为记录已经写入的 redo 日志量，设计了一个称之为 Log Sequeue Number 的全局变量，翻译过来就是： 日志序列号 ，简称 lsn 。不过不像人一出生的年龄是 0 岁，设计 InnoDB 的大叔规定初始的 lsn 值为 8704 （也就是一条 redo 日志也没写入时， lsn 的值为 8704 ）。

我们知道在向 log buffer 中写入 redo 日志时不是一条一条写入的，而是以一个 mtr 生成的一组 redo 日志为单位进行写入的。而且实际上是把日志内容写在了 log block body 处。但是在统计 lsn 的增长量时，是按照实际写入的日志量加上占用的 log block header 和 log block trailer 来计算的。我们来看一个例子：

- 系统第一次启动后初始化 log buffer 时， buf_free （就是标记下一条 redo 日志应该写入到 log buffer 的位置的变量）就会指向第一个 block 的偏移量为12字节（ log block header 的大小）的地方，那么 lsn 值也会跟着增加12：

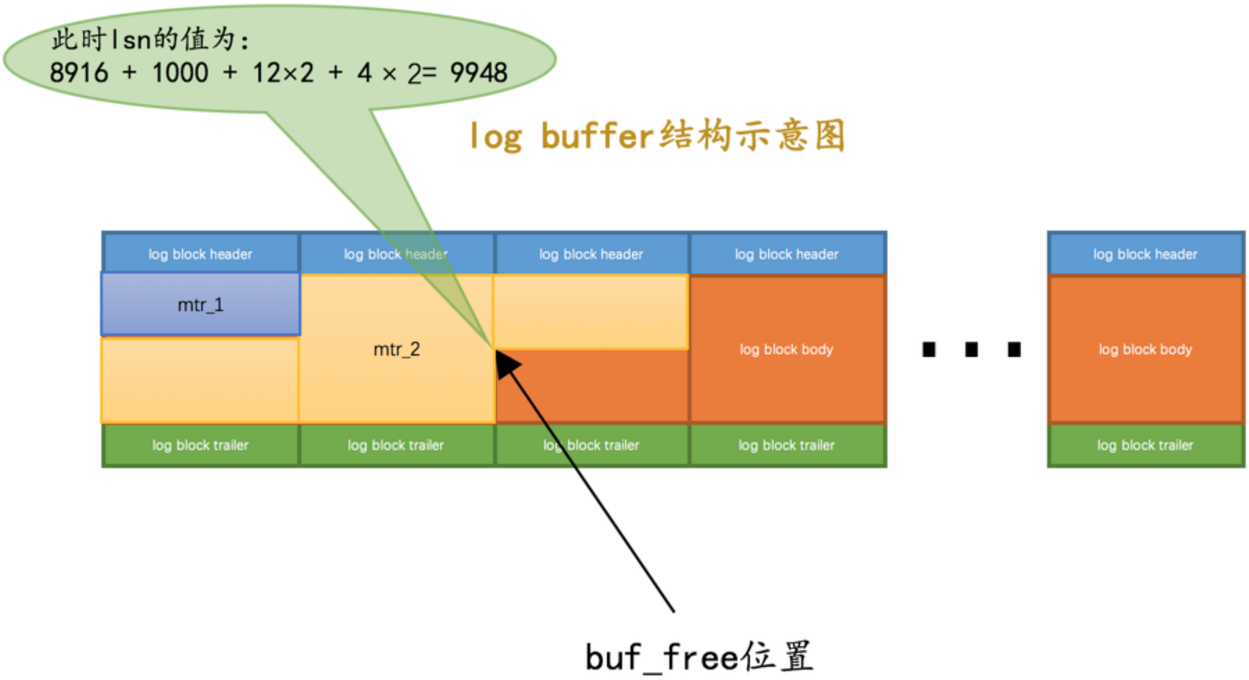


- 如果某个 mtr 产生的一组 redo 日志占用的存储空间比较小，也就是待插入的block剩余空闲空间能容纳这个 mtr 提交的日志时， lsn 增长的量就是该 mtr 生成的 redo 日志占用的字节数，就像这样：



我们假设上图中 mtr_1 产生的 redo 日志量为200字节，那么 lsn 就要在 8716 的基础上增加 200 ，变为 8916 。

- 如果某个 mtr 产生的一组 redo 日志占用的存储空间比较大，也就是待插入的block剩余空闲空间不足以容纳这个 mtr 提交的日志时， lsn 增长的量就是该 mtr 生成的 redo 日志占用的字节数加上额外占用的 log block header 和 log block trailer 的字节数，就像这样：



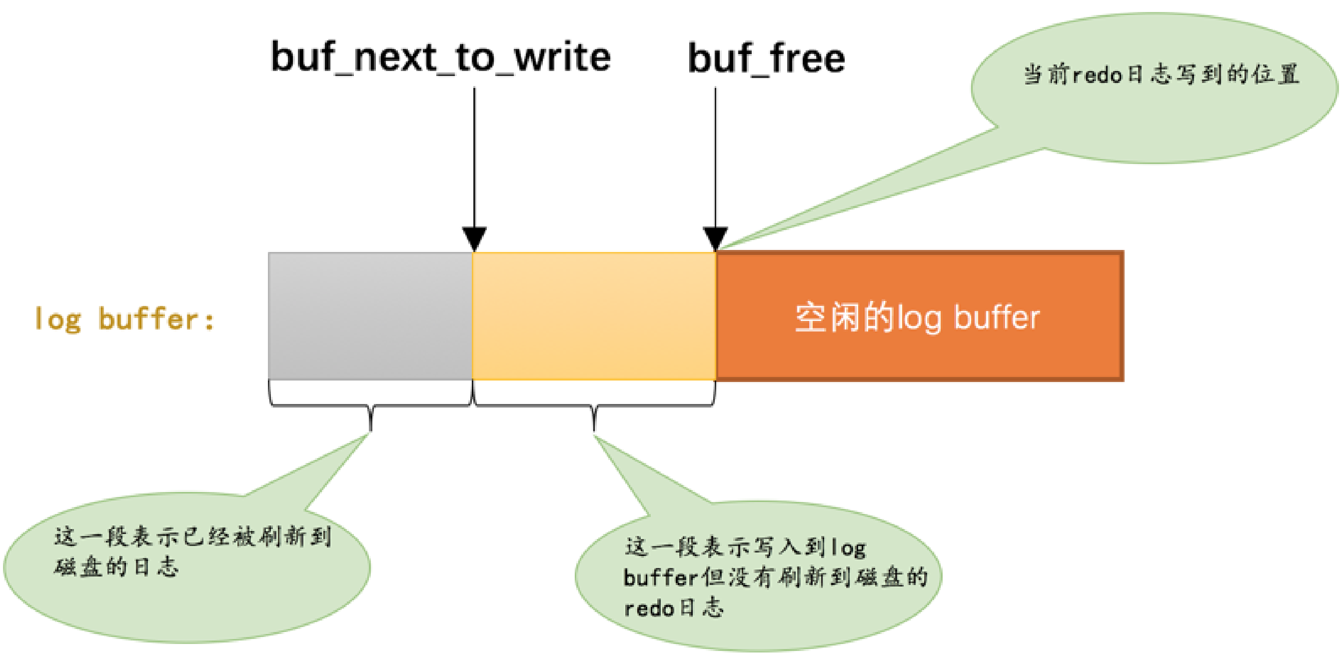
我们假设上图中 mtr_2 产生的 redo 日志量为1000字节，为了将 mtr_2 产生的 redo 日志写入 log buffer ，我们不得不额外多分配两个block，所以 lsn 的值需要在 8916 的基础上增加 1000 + 12×2 + 4 × 2 = 1032 。

小贴士：
为什么初始的lsn值为8704呢？我也不太清楚，人家就这么规定的。其实你也可以规定你一生下来算1岁，只要保证随着时间的流逝，你的年龄不断增长就好了。

从上边的描述中可以看出来，每一组由mtr生成的redo日志都有一个唯一的LSN值与其对应，LSN值越小，说明redo日志产生的越早。

21.2.1 flushed_to_disk_lsn

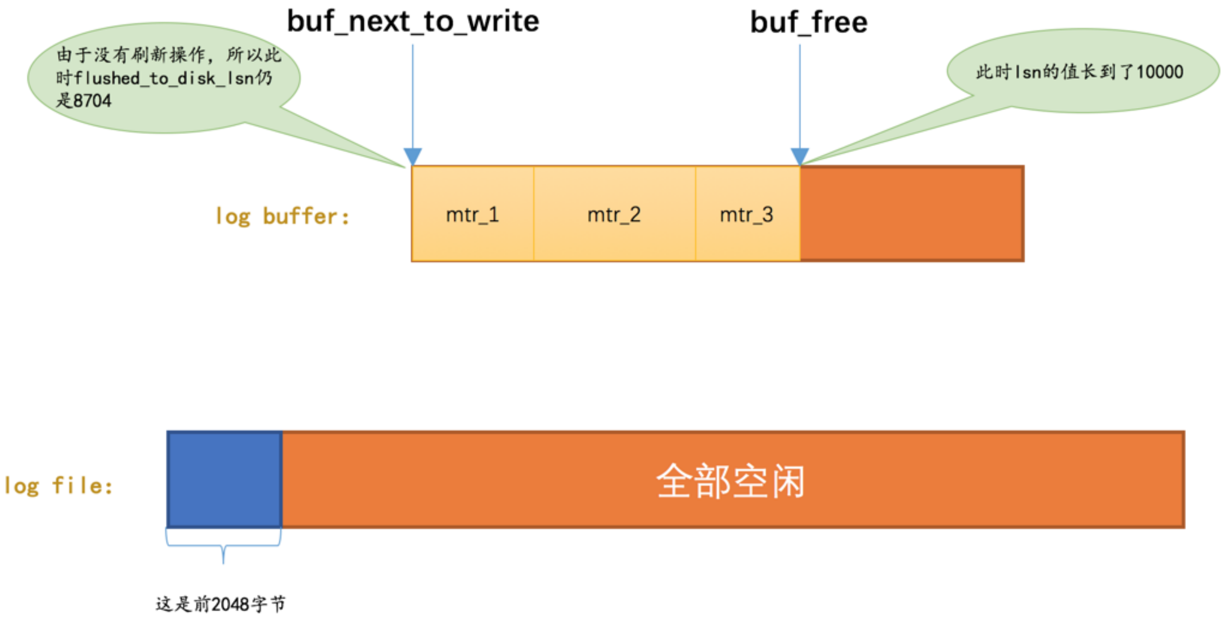
redo 日志是首先写到 log buffer 中，之后才会被刷新到磁盘上的 redo 日志文件。所以设计 InnoDB 的大叔提出了一个称之为 buf_next_to_write 的全局变量，标记当前 log buffer 中已经有哪些日志被刷新到磁盘中了。画个图表示就是这样：



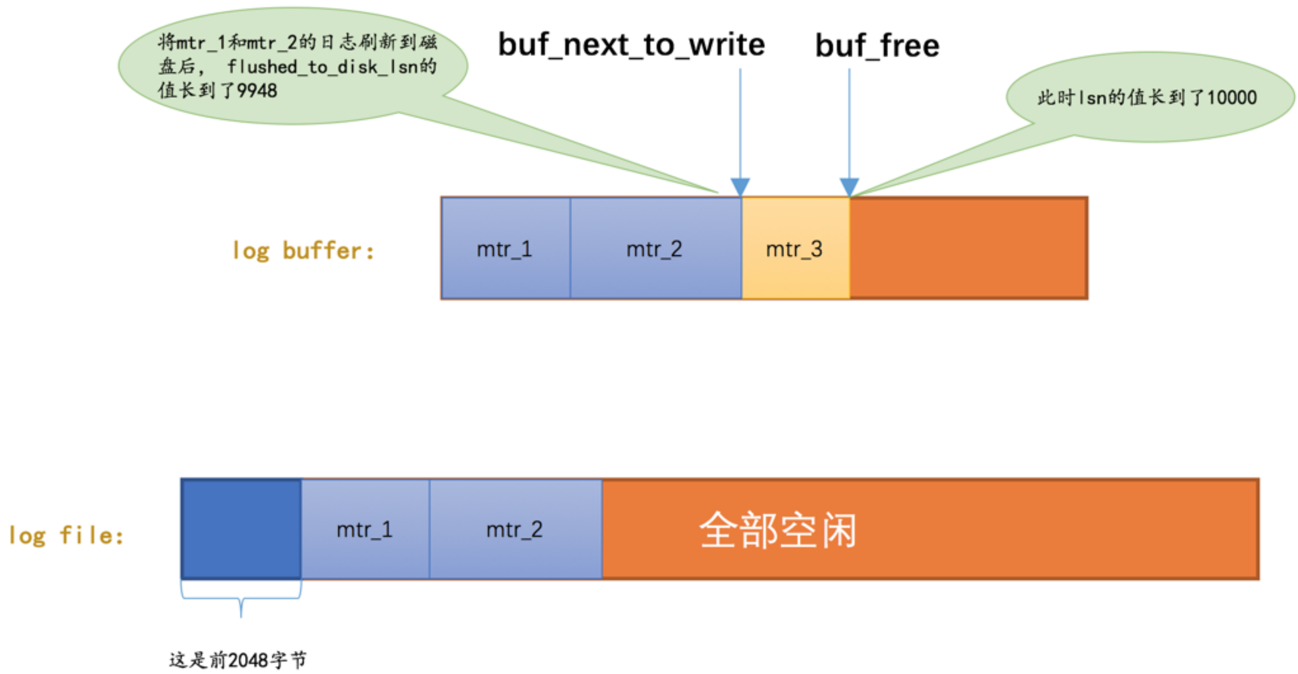
我们前边说 lsn 是表示当前系统中写入的 redo 日志量，这包括了写到 log buffer 而没有刷新到磁盘的日志，相应的，设计 InnoDB 的大叔提出了一个表示刷新到磁盘中的 redo 日志量的全局变量，称之为 flushed_to_disk_lsn 。系统第一次启动时，该变量的值和初始的 lsn 值是相同的，都是 8704 。随着系统的运行， redo 日志被不断写入 log buffer ，但是并不会立即刷新到磁盘， lsn 的值就和 flushed_to_disk_lsn 的值拉开了差距。我们演示一下：

- 系统第一次启动后，向 log buffer 中写入了 mtr_1 、 mtr_2 、 mtr_3 这三个 mtr 产生的 redo 日志，假设这三个 mtr 开始和结束时对应的lsn值分别是：
  - mtr_1 ：8716 ~ 8916
  - mtr_2 ：8916 ~ 9948
  - mtr_3 ：9948 ~ 10000

  此时的 lsn 已经增长到了10000，但是由于没有刷新操作，所以此时 flushed_to_disk_lsn 的值仍为 8704 ，如图：



- 随后进行将 log buffer 中的block刷新到 redo 日志文件的操作，假设将 mtr_1 和 mtr_2 的日志刷新到磁盘，那么 flushed_to_disk_lsn 就应该增长 mtr_1 和 mtr_2 写入的日志量，所以 flushed_to_disk_lsn 的值增长到了 9948 ，如图：

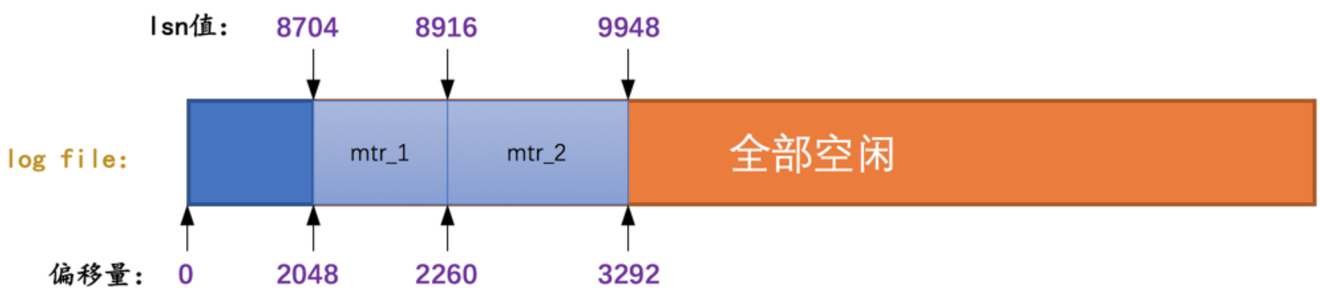


综上所述，当有新的 redo 日志写入到 log buffer 时，首先 lsn 的值会增长，但 flushed_to_disk_lsn 不变，随后随着不断有 log buffer 中的日志被刷新到磁盘上， flushed_to_disk_lsn 的值也跟着增长。如果两者的值相同时，说明log buffer中的所有redo日志都已经刷新到磁盘中了。

> 小贴士：
> 应用程序向磁盘写入文件时其实是先写到操作系统的缓冲区中去，如果某个写入操作要等到操作系统确认已经写到磁盘时才返回，那需要调用一下操作系统提供的fsync函数。其实只有当系统执行了fsync函数后，flushed_to_disk_lsn的值才会跟着增长，当仅仅把log buffer中的日志写入到操作系统缓冲区却没有显式的刷新到磁盘时，另外的一个称之为write_lsn的值跟着增长。不过为了大家理解上的方便，我们在讲述时把flushed_to_disk_lsn和write_lsn的概念混淆了起来。

21.2.2 lsn值和redo日志文件偏移量的对应关系

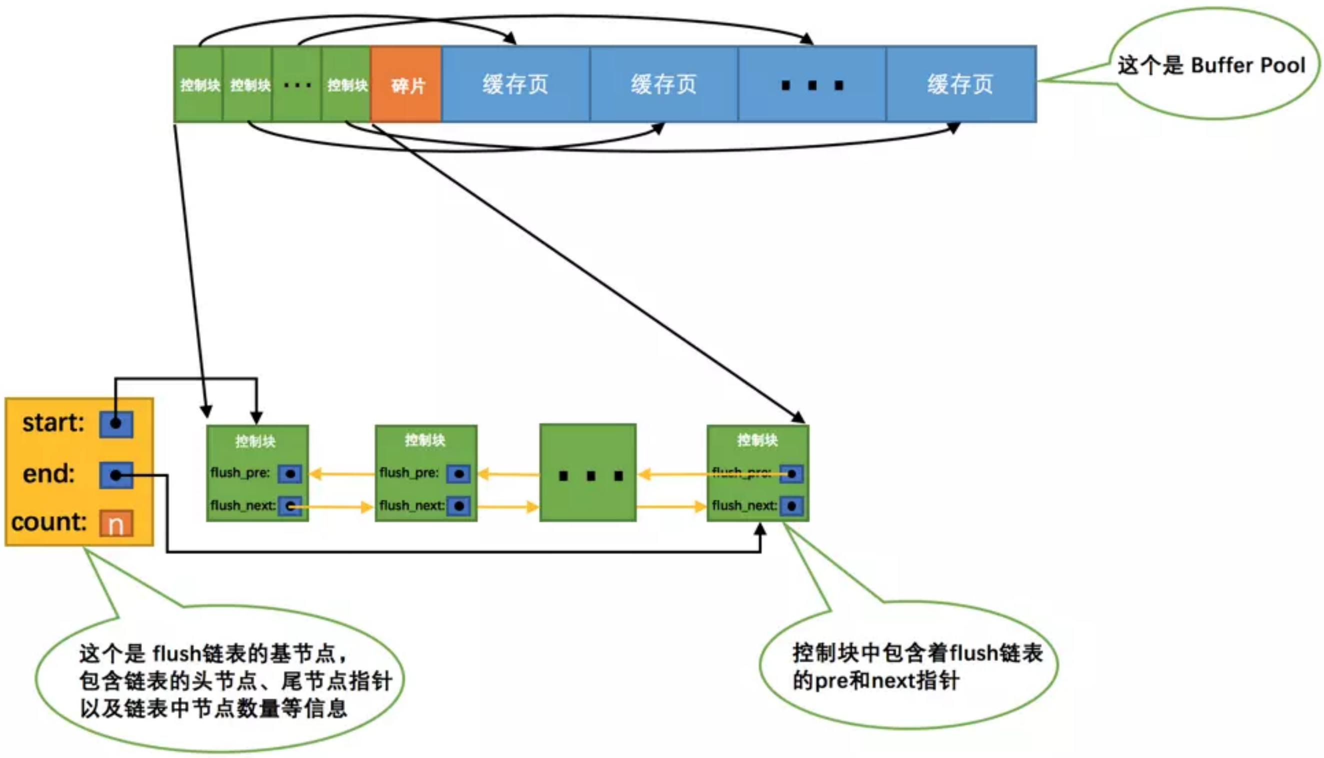
因为 lsn 的值是代表系统写入的 redo 日志量的一个总和，一个 mtr 中产生多少日志， lsn 的值就增加多少（当然有时候要加上 log block header 和 log block trailer 的大小），这样 mtr 产生的日志写到磁盘中时，很容易计算某一个 lsn 值在 redo 日志文件组中的偏移量，如图：



初始时的 LSN 值是 8704 ，对应文件偏移量 2048 ，之后每个 mtr 向磁盘中写入多少字节日志， lsn 的值就增长多少。

21.2.3 flush链表中的LSN

我们知道一个 mtr 代表一次对底层页面的原子访问，在访问过程中可能会产生一组不可分割的 redo 日志，在 mtr 结束时，会把这一组 redo 日志写入到 log buffer 中。除此之外，在 mtr 结束时还有一件非常重要的事情要做，就是把在mtr执行过程中可能修改过的页面加入到Buffer Pool的flush链表。为了防止大家早已忘记 flush链表 是个啥，我们再看一下图：

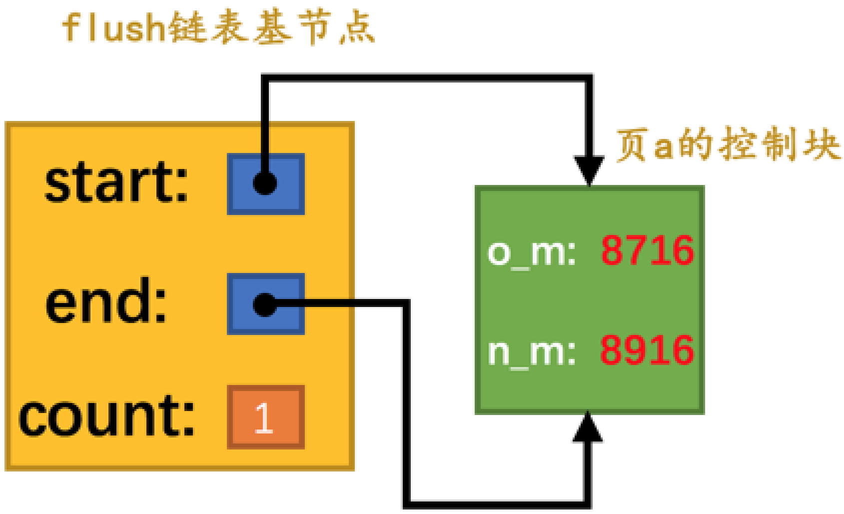


当第一次修改某个缓存在 Buffer Pool 中的页面时，就会把这个页面对应的控制块插入到 flush链表 的头部，之后再修改该页面时由于它已经在 flush 链表中了，就不再次插入了。也就是说flush链表中的脏页是按照页面的第一次修改时间从大到小进行排序的。在这个过程中会在缓存页对应的控制块中记录两个关于页面何时修改的属性：

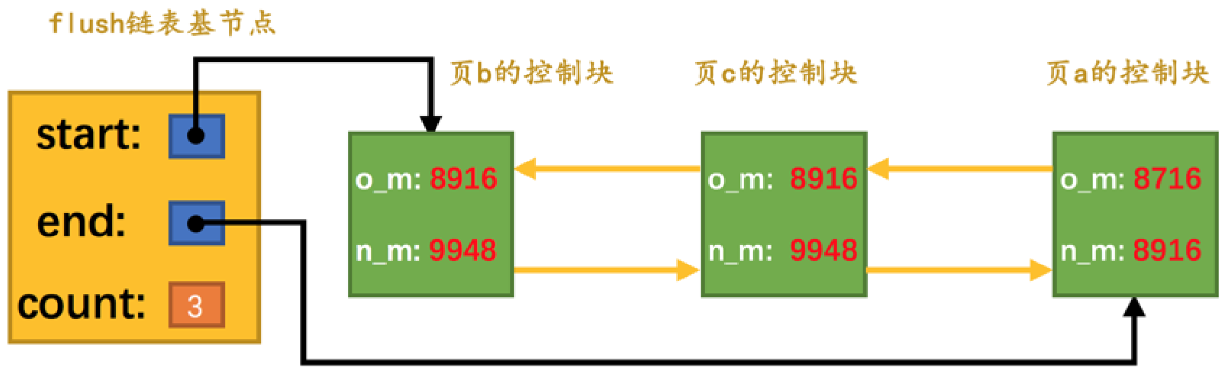
- oldest_modification ：如果某个页面被加载到 Buffer Pool 后进行第一次修改，那么就将修改该页面的 mtr 开始时对应的 lsn 值写入这个属性。
- newest_modification ：每修改一次页面，都会将修改该页面的 mtr 结束时对应的 lsn 值写入这个属性。也就是说该属性表示页面最近一次修改后对应的系统 lsn 值。

我们接着上边唠叨 flushed_to_disk_lsn 的例子看一下：

- 假设 mtr_1 执行过程中修改了 页a ，那么在 mtr_1 执行结束时，就会将 页a 对应的控制块加入到 flush链表 的头部。并且将 mtr_1 开始时对应的 lsn ，也就是 8716 写入 页a 对应的控制块的 oldest_modification 属性中，把 mtr_1 结束时对应的 lsn ，也就是8916写入 页a 对应的控制块的 newest_modification 属性中。画个图表示一下（为了让图片美观一些，我们把 oldest_modification 缩写成了 o_m ，把 newest_modification 缩写成了 n_m ）：

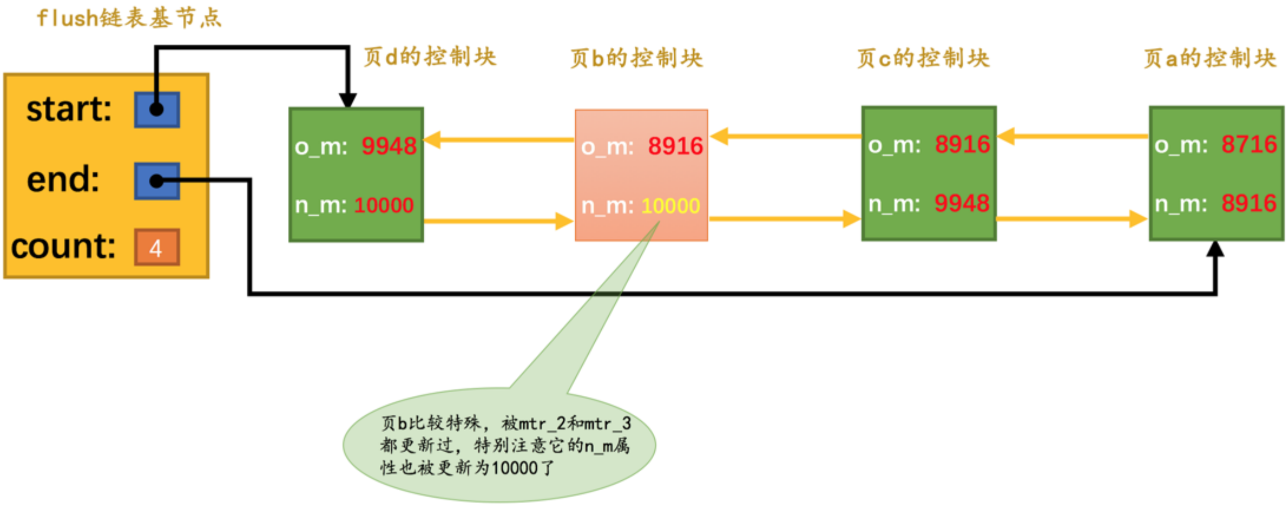


- 接着假设 mtr_2 执行过程中又修改了 页b 和 页c 两个页面，那么在 mtr_2 执行结束时，就会将 页b 和 页c 对应的控制块都加入到 flush链表 的头部。并且将 mtr_2 开始时对应的 lsn ，也就是8916写入 页b 和 页c 对应的控制块的 oldest_modification 属性中，把 mtr_2 结束时对应的 lsn ，也就是9948写入 页b 和 页c 对应的控制块的 newest_modification 属性中。画个图表示一下：



从图中可以看出来，每次新插入到 flush链表 中的节点都是被放在了头部，也就是说 flush链表 中前边的脏页修改的时间比较晚，后边的脏页修改时间比较早。

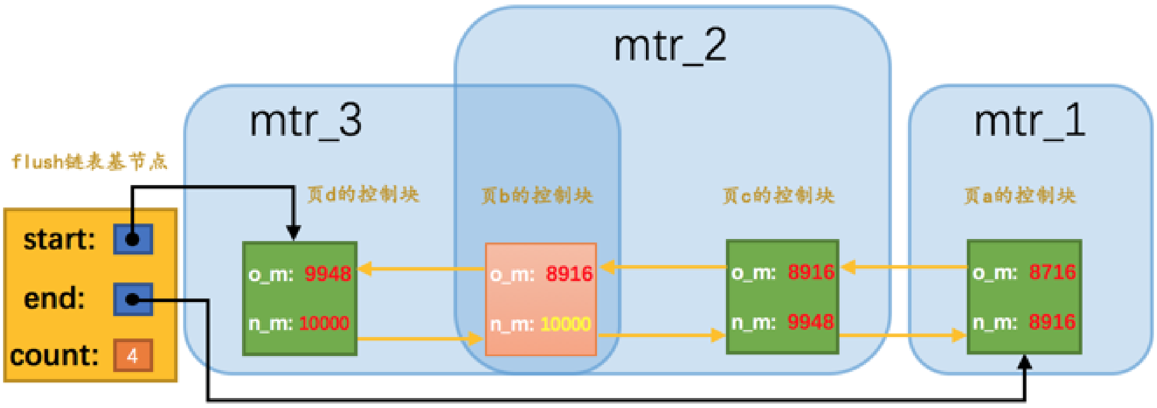
- 接着假设 mtr_3 执行过程中修改了 页b 和 页d ，不过 页b 之前已经被修改过了，所以它对应的控制块已经被插入到了 flush 链表，所以在 mtr_3 执行结束时，只需要将 页d 对应的控制块都加入到 flush链表 的头部即可。所以需要将 mtr_3 开始时对应的 lsn ，也就是9948写入 页d 对应的控制块的 oldest_modification 属性中，把 mtr_3 结束时对应的 lsn ，也就是10000写入 页d 对应的控制块的 newest_modification 属性中。另外，由于 页b 在 mtr_3 执行过程中又发生了一次修改，所以需要更新 页b 对应的控制块中 newest_modification 的值为10000。画个图表示一下：

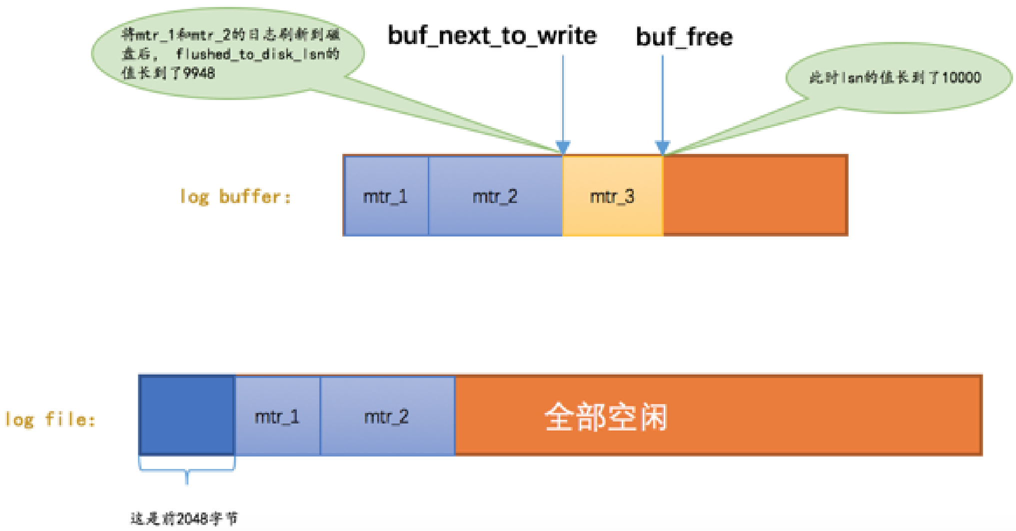


总结一下上边说的，就是：flush链表中的脏页按照修改发生的时间顺序进行排序，也就是按照oldest_modification代表的LSN值进行排序，被多次更新的页面不会重复插入到flush链表中，但是会更新newest_modification属性的值。

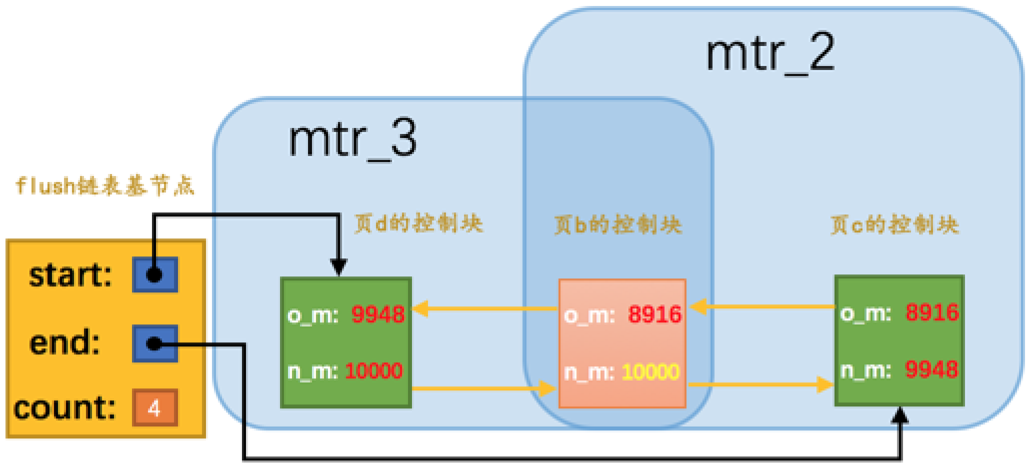
21.3 checkpoint

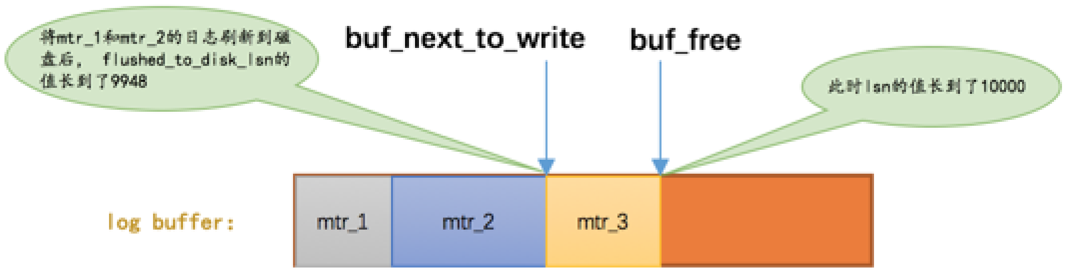
有一个很不幸的事实就是我们的 redo 日志文件组容量是有限的，我们不得不选择循环使用 redo 日志文件组中的文件，但是这会造成最后写的 redo 日志与最开始写的 redo 日志 追尾 ，这时应该想到：redo日志只是为了系统奔溃后恢复脏页用的，如果对应的脏页已经刷新到了磁盘，也就是说即使现在系统奔溃，那么在重启后也用不着使用redo日志恢复该页面了，所以该redo日志也就没有存在的必要了，那么它占用的磁盘空间就可以被后续的redo日志所重用。也就是说：判断某些redo日志占用的磁盘空间是否可以覆盖的依据就是它对应的脏页是否已经刷新到磁盘里。我们看一下前边一直唠叨的那个例子：





如图，虽然 mtr_1 和 mtr_2 生成的 redo 日志都已经被写到了磁盘上，但是它们修改的脏页仍然留在 Buffer Pool 中，所以它们生成的 redo 日志在磁盘上的空间是不可以被覆盖的。之后随着系统的运行，如果 页a 被刷新到了磁盘，那么它对应的控制块就会从 flush链表 中移除，就像这样子：







这样 mtr_1 生成的 redo 日志就没有用了，它们占用的磁盘空间就可以被覆盖掉了。设计 InnoDB 的大叔提出了一个全局变量 checkpoint_lsn 来代表当前系统中可以被覆盖的 redo 日志总量是多少，这个变量初始值也是 8704 。

比方说现在 页a 被刷新到了磁盘， mtr_1 生成的 redo 日志就可以被覆盖了，所以我们可以进行一个增加 checkpoint_lsn 的操作，我们把这个过程称之为做一次 checkpoint 。做一次 checkpoint 其实可以分为两个步骤：

- 步骤一：计算一下当前系统中可以被覆盖的 redo 日志对应的 lsn 值最大是多少。

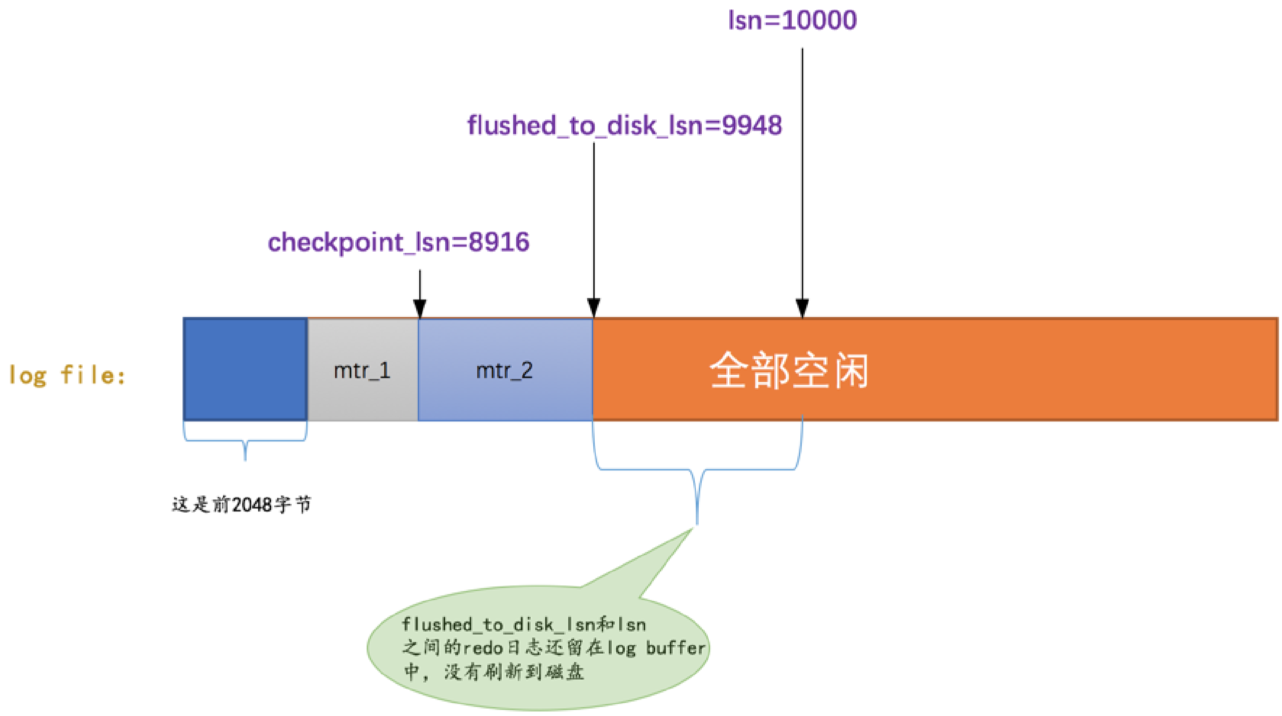
  redo 日志可以被覆盖，意味着它对应的脏页被刷到了磁盘，只要我们计算出当前系统中被最早修改的脏页对应的 oldest_modification 值，那凡是在系统lsn值小于该节点的oldest_modification值时产生的redo日志都是可以被覆盖掉的，我们就把该脏页的 oldest_modification 赋值给 checkpoint_lsn 。

  比方说当前系统中 页a 已经被刷新到磁盘，那么 flush链表 的尾节点就是 页c ，该节点就是当前系统中最早修改的脏页了，它的 oldest_modification 值为8916，我们就把8916赋值给 checkpoint_lsn （也就是说在redo日志对应的lsn值小于8916时就可以被覆盖掉）。
- 步骤二：将 checkpoint_lsn 和对应的 redo 日志文件组偏移量以及此次 checkpint 的编号写到日志文件的管理信息（就是 checkpoint1 或者 checkpoint2 ）中。

设计 InnoDB 的大叔维护了一个目前系统做了多少次 checkpoint 的变量 checkpoint_no ，每做一次 checkpoint ，该变量的值就加1。我们前边说过计算一个 lsn 值对应的 redo 日志文件组偏移量是很容易的，所以可以计算得到该 checkpoint_lsn 在 redo 日志文件组中对应的偏移量 checkpoint_offset ，然后把这三个值都写到 redo 日志文件组的管理信息中。

我们说过，每一个 redo 日志文件都有 2048 个字节的管理信息，但是上述关于checkpoint的信息只会被写到日志文件组的第一个日志文件的管理信息中。不过我们是存储到 checkpoint1 中还是 checkpoint2 中呢？设计 InnoDB 的大叔规定，当 checkpoint_no 的值是偶数时，就写到 checkpoint1 中，是奇数时，就写到 checkpoint2 中。

记录完 checkpoint 的信息之后， redo 日志文件组中各个 lsn 值的关系就像这样：



21.3.1 批量从flush链表中刷出脏页

我们在介绍 Buffer Pool 的时候说过，一般情况下都是后台的线程在对 LRU链表 和 flush链表 进行刷脏操作，这主要因为刷脏操作比较慢，不想影响用户线程处理请求。但是如果当前系统修改页面的操作十分频繁，这样就导致写日志操作十分频繁，系统 lsn 值增长过快。如果后台的刷脏操作不能将脏页刷出，那么系统无法及时做 checkpoint ，可能就需要用户线程同步的从 flush链表 中把那些最早修改的脏页（ oldest_modification 最小的脏页）刷新到磁盘，这样这些脏页对应的 redo 日志就没用了，然后就可以去做 checkpoint 了。

21.3.2 查看系统中的各种LSN值

我们可以使用 SHOW ENGINE INNODB STATUS 命令查看当前 InnoDB 存储引擎中的各种 LSN 值的情况，比如：

```
mysql> SHOW ENGINE INNODB STATUS\G

(...省略前边的许多状态)
LOG
---
Log sequence number 124476971
Log flushed up to   124099769
Pages flushed up to 124052503
Last checkpoint at  124052494
0 pending log flushes, 0 pending chkp writes
24 log i/o's done, 2.00 log i/o's/second
----------------------
(...省略后边的许多状态)
```

其中：

- `Log sequence number`：代表系统中的 `lsn` 值，也就是当前系统已经写入的 `redo` 日志量，包括写入 `log buffer` 中的日志。
- `Log flushed up to`：代表 `flushed_to_disk_lsn` 的值，也就是当前系统已经写入磁盘的 `redo` 日志量。
- `Pages flushed up to`：代表 `flush链表` 中被最早修改的那个页面对应的 `oldest_modification` 属性值。
- `Last checkpoint at`：当前系统的 `checkpoint_lsn` 值。

21.4 innodb_flush_log_at_trx_commit的用法

我们前边说为了保证事务的 `持久性`，用户线程在事务提交时需要将该事务执行过程中产生的所有 `redo` 日志都刷新到磁盘上。这一条要求太狠了，会很明显的降低数据库性能。如果有的同学对事务的 `持久性` 要求不是那么强烈的话，可以选择修改一个称为 `innodb_flush_log_at_trx_commit` 的系统变量的值，该变量有3个可选的值：

- `0`：当该系统变量值为0时，表示在事务提交时不立即向磁盘中同步 `redo` 日志，这个任务是交给后台线程做的。

  这样很明显会加快请求处理速度，但是如果事务提交后服务器挂了，后台线程没有及时将 `redo` 日志刷新到磁盘，那么该事务对页面的修改会丢失。
- `1`：当该系统变量值为1时，表示在事务提交时需要将 `redo` 日志同步到磁盘，可以保证事务的 `持久性`。`1` 也是 `innodb_flush_log_at_trx_commit` 的默认值。
- `2`：当该系统变量值为2时，表示在事务提交时需要将 `redo` 日志写到操作系统的缓冲区中，但并不需要保证将日志真正的刷新到磁盘。

  这种情况下如果数据库挂了，操作系统没挂的话，事务的 `持久性` 还是可以保证的，但是操作系统也挂了的话，那就不能保证 `持久性` 了。

21.5 崩溃恢复

在服务器不挂的情况下，`redo` 日志简直就是个大累赘，不仅没用，反而让性能变得更差。但是万一，我说万一啊，万一数据库挂了，那 `redo` 日志可是个宝了，我们就可以在重启时根据 `redo` 日志中的记录就可以将页面恢复到系统奔溃前的状态。我们接下来大致看一下恢复过程是个啥样。

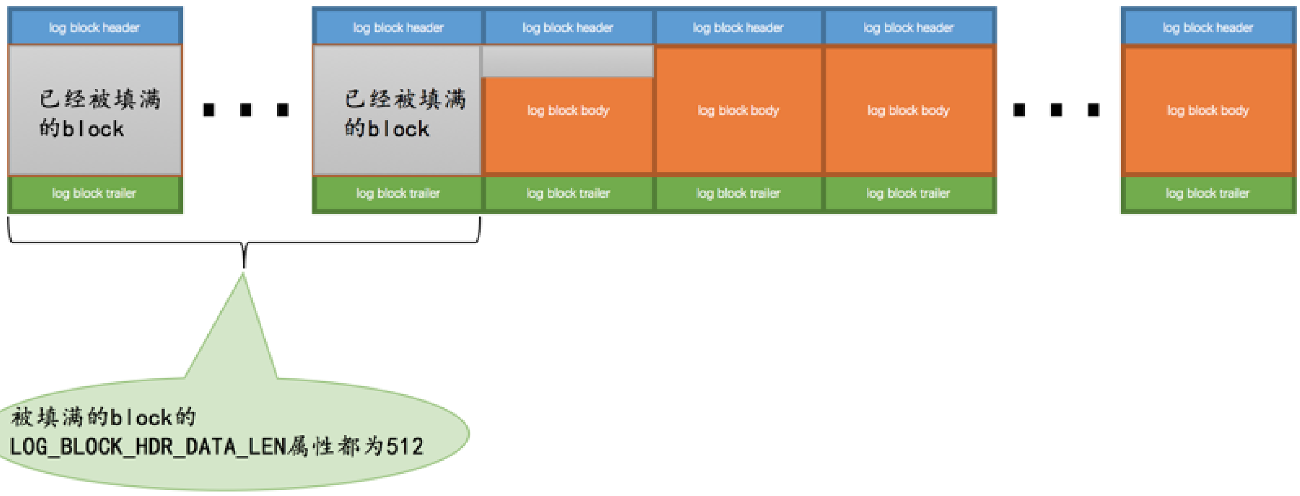
21.5.1 确定恢复的起点

我们前边说过，`checkpoint_lsn` 之前的 `redo` 日志都可以被覆盖，也就是说这些 `redo` 日志对应的脏页都已经被刷新到磁盘中了，既然它们已经被刷盘，我们就没必要恢复它们了。对于 `checkpoint_lsn` 之后的 `redo` 日志，它们对应的脏页可能没被刷盘，也可能被刷盘了，我们不能确定，所以需要从 `checkpoint_lsn` 开始读取 `redo` 日志来恢复页面。

当然， redo 日志文件组的第一个文件的管理信息中有两个block都存储了 checkpoint_lsn 的信息，我们当然是要选取最近发生的那次checkpoint的信息。衡量 checkpoint 发生时间早晚的信息就是所谓的 checkpoint_no ，我们只要把 checkpoint1 和 checkpoint2 这两个block中的 checkpoint_no 值读出来比一下大小，哪个的 checkpoint_no 值更大，说明哪个block存储的就是最近的一次 checkpoint 信息。这样我们就能拿到最近发生的 checkpoint 对应的 checkpoint_lsn 值以及它在 redo 日志文件组中的偏移量 checkpoint_offset 。

21.5.2 确定恢复的终点

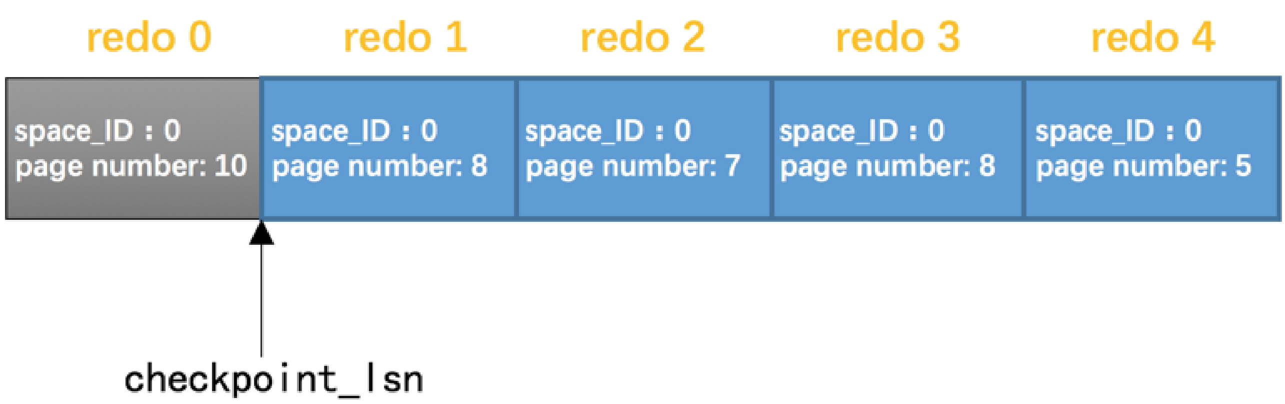
redo 日志恢复的起点确定了，那终点是哪个呢？这个还得从block的结构说起。我们说在写 redo 日志的时候都是顺序写的，写满了一个block之后会再往下一个block中写：



普通block的 log block header 部分有一个称之为 LOG_BLOCK_HDR_DATA_LEN 的属性，该属性值记录了当前block里使用了多少字节的空间。对于被填满的block来说，该值永远为 512 。如果该属性的值不为 512 ，那么就是它了，它就是此次奔溃恢复中需要扫描的最后一个block。

21.5.3 怎么恢复

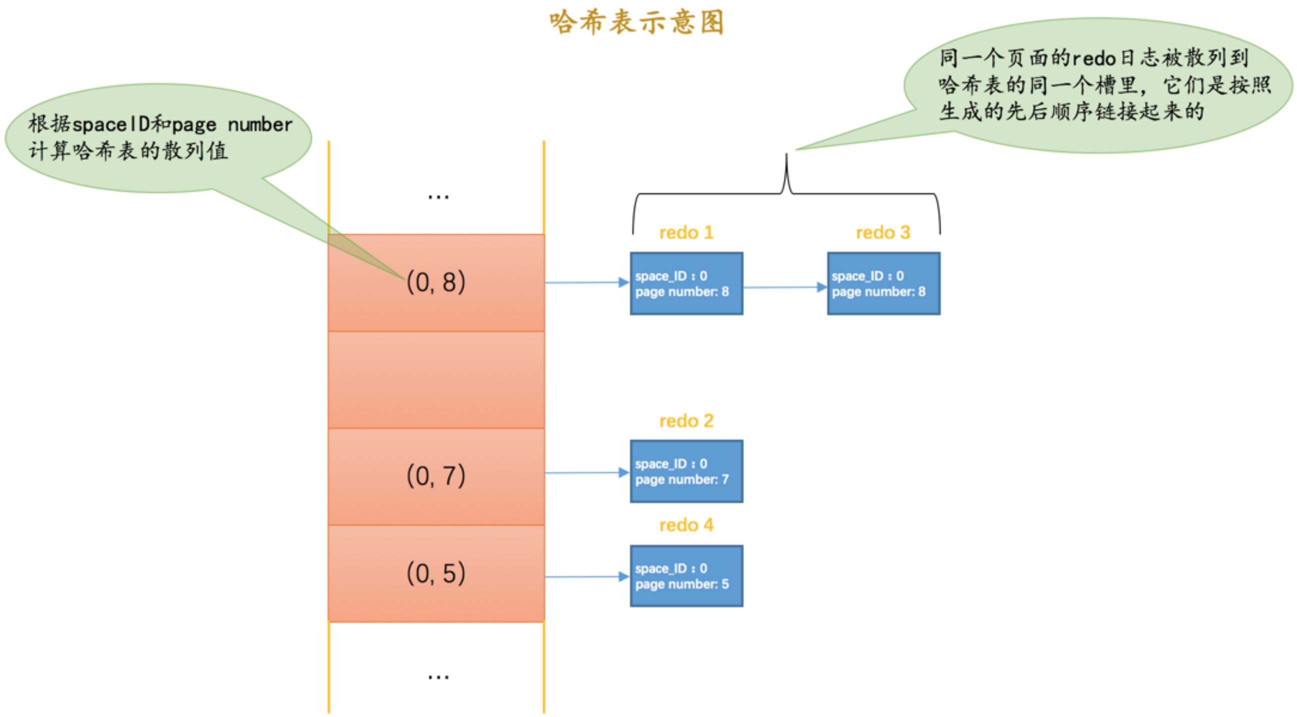
确定了需要扫描哪些 redo 日志进行奔溃恢复之后，接下来就是怎么进行恢复了。假设现在的 redo 日志文件中有5条 redo 日志，如图：



由于 redo 0 在 checkpoint_lsn 后边，恢复时可以不管它。我们现在可以按照 redo 日志的顺序依次扫描 checkpoint_lsn 之后的各条redo日志，按照日志中记载的内容将对应的页面恢复出来。这样没什么问题，不过设计 InnoDB 的大叔还是想了一些办法加快这个恢复的过程：

- 使用哈希表

根据 redo 日志的 space ID 和 page number 属性计算出散列值，把 space ID 和 page number 相同的 redo 日志放到哈希表的同一个槽里，如果有多个 space ID 和 page number 都相同的 redo 日志，那么它们之间使用链表连接起来，按照生成的先后顺序链接起来的，如图所示：



之后就可以遍历哈希表，因为对同一个页面进行修改的 redo 日志都放在了一个槽里，所以可以一次性将一个页面修复好（避免了很多读取页面的随机IO），这样可以加快恢复速度。另外需要注意一点的是，同一个页面的 redo 日志是按照生成时间顺序进行排序的，所以恢复的时候也是按照这个顺序进行恢复，如果不按照生成时间顺序进行排序的话，那么可能出现错误。比如原先的修改操作是先插入一条记录，再删除该条记录，如果恢复时不按照这个顺序来，就可能变成先删除一条记录，再插入一条记录，这显然是错误的。

- 跳过已经刷新到磁盘的页面

  我们前边说过， checkpoint_lsn 之前的 redo 日志对应的脏页确定都已经刷到磁盘了，但是 checkpoint_lsn 之后的 redo 日志我们不能确定是否已经刷到磁盘，主要是因为在最近做的一次 checkpoint 后，可能后台线程又不断的从 LRU链表 和 flush链表 中将一些脏页刷出 Buffer Pool 。这些在 checkpoint_lsn 之后的 redo 日志，如果它们对应的脏页在奔溃发生时已经刷新到磁盘，那在恢复时也就没有必要根据 redo 日志的内容修改该页面了。

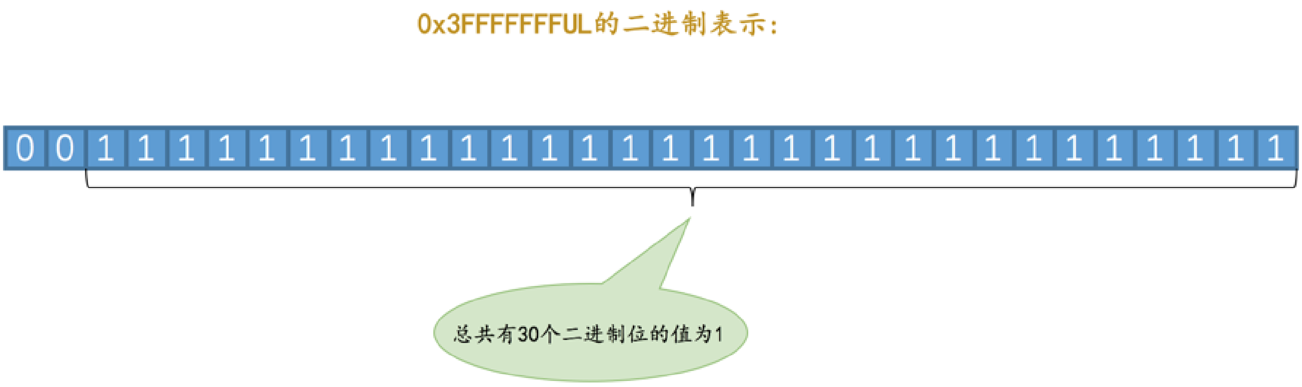
  那在恢复时怎么知道某个 redo 日志对应的脏页是否在奔溃发生时已经刷新到磁盘了呢？这还得从页面的结构说起，我们前边说过每个页面都有一个称之为 File Header 的部分，在 File Header 里有一个称之为 FIL_PAGE_LSN 的属性，该属性记载了最近一次修改页面时对应的 lsn 值（其实就是页面控制块中的 newest_modification 值）。如果在做了某次 checkpoint 之后有脏页被刷新到磁盘中，那么该页对应的 FIL_PAGE_LSN 代表的 lsn 值肯定大于 checkpoint_lsn 的值，凡是符合这种情况的页面就不需要重复执行lsn值小于 FIL_PAGE_LSN 的redo日志了，所以更进一步提升了奔溃恢复的速度。

21.6 遗漏的问题：LOG_BLOCK_HDR_NO是如何计算的

我们前边说过，对于实际存储 redo 日志的普通的 log block 来说，在 log block header 处有一个称之为 LOG_BLOCK_HDR_NO 的属性（忘记了的话回头再看看哈），我们说这个属性代表一个唯一的标号。这个属性是初次使用该block时分配的，跟当时的系统 lsn 值有关。使用下边的公式计算该block的 LOG_BLOCK_HDR_NO 值：

```
((lsn / 512) & 0x3FFFFFFFUL) + 1
```

这个公式里的 0x3FFFFFFFUL 可能让大家有点困惑，其实它的二进制表示可能更亲切一点：



从图中可以看出， 0x3FFFFFFFUL 对应的二进制数的前2位为0，后30位的值都为 1 。我们刚开始学计算机的时候就学过，一个二进制位与0做与运算（ & ）的结果肯定是0，一个二进制位与1做与运算（ & ）的结果就是原值。让一个数和 0x3FFFFFFFUL 做与运算的意思就是要将该值的前2个比特位的值置为0，这样该值就肯定小于或等于 0x3FFFFFFFUL 了。这也就说明了，不论lsn多大， ((lsn / 512) & 0x3FFFFFFFUL) 的值肯定在 0 ~~0x3FFFFFFFUL 之间，再加1的话肯定在 1~~ 0x40000000UL 之间。而 0x40000000UL 这个值大家应该很熟悉，这个值就代表着 1GB 。也就是说系统最多能产生不重复的 LOG_BLOCK_HDR_NO 值只有 1GB 个。设计InnoDB的大叔规定 redo 日志文件组中包含的所有文件大小总和不得超过512GB，一个block大小是512字节，也就是说redo日志文件组中包含的block块最多为1GB个，所以有1GB个不重复的编号值也就够用了。

另外， LOG_BLOCK_HDR_NO 值的第一个比特位比较特殊，称之为 flush bit ，如果该值为1，代表着本block是在某次将 log buffer 中的block刷新到磁盘的操作中的第一个被刷入的block。

22 第22章 后悔了怎么办-undo日志（上）

标签： MySQL是怎样运行的

22.1 事务回滚的需求

我们说过 事务 需要保证 原子性 ，也就是事务中的操作要么全部完成，要么什么也不做。但是偏偏有时候事务执行到一半会出现一些情况，比如：

- 情况一：事务执行过程中可能遇到各种错误，比如服务器本身的错误，操作系统错误，甚至是突然断电导致的错误。
- 情况二：程序员可以在事务执行过程中手动输入 ROLLBACK 语句结束当前的事务的执行。

这两种情况都会导致事务执行到一半就结束，但是事务执行过程中可能已经修改了很多东西，为了保证事务的原子性，我们需要把东西改回原先的样子，这个过程就称之为 回滚 （英文名： rollback ），这样就可以造成一个假象：这个事务看起来什么都没做，所以符合 原子性 要求。

小时候我非常痴迷于象棋，总是想找厉害的大人下棋，赢棋是不可能赢棋的，这辈子都不可能赢棋的，又不想认输，只能偷偷的悔棋才能勉强玩的下去。 悔棋 就是一种非常典型的 回滚 操作，比如棋子往前走两步， 悔棋 对应的操作就是向后走两步；比如棋子往左走一步， 悔棋 对应的操作就是向右走一步。数据库中的回滚跟 悔棋 差不多，你插入了一条记录， 回滚 操作对应的就是把这条记录删除掉；你更新了一条记录， 回滚 操作对应的就是把该记录更新为旧值；你删除了一条记录， 回滚 操作对应的自然就是把该记录再插进去。说的貌似很简单的样子[手动偷笑😏]。

从上边的描述中我们已经能隐约感觉到，每当我们要对一条记录做改动时（这里的 改动 可以指 INSERT 、 DELETE 、 UPDATE ），都需要留一手 —— 把回滚时所需的东西都给记下来。比方说：

- 你插入一条记录时，至少要把这条记录的主键值记下来，之后回滚的时候只需要把这个主键值对应的记录删掉就好了。
- 你删除了一条记录，至少要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中就好了。
- 你修改了一条记录，至少要把修改这条记录前的旧值都记录下来，这样之后回滚时再把这条记录更新为旧值就好了。

设计数据库的大叔把这些为了回滚而记录的这些东东称之为撤销日志，英文名为 `undo log` ，我们也可以土洋结合，称之为 `undo日志` 。这里需要注意的一点是，由于查询操作（ `SELECT` ）并不会修改任何用户记录，所以在查询操作执行时，并不需要记录相应的 `undo日志` 。在真实的 `InnoDB` 中， `undo日志` 其实并不像我们上边所说的那么简单，不同类型的操作产生的 `undo日志` 的格式也是不同的，不过先暂时把这些容易让人脑子糊的具体细节放一放，我们先回过头来看看 `事务id` 是个神马玩意儿。

22.2 事务id

22.2.1 给事务分配id的时机

我们前边在唠叨 `事务简介` 时说过，一个事务可以是一个只读事务，或者是一个读写事务：

- 我们可以通过 `START TRANSACTION READ ONLY` 语句开启一个只读事务。

  在只读事务中不可以对普通的表（其他事务也能访问到的表）进行增、删、改操作，但可以对临时表做增、删、改操作。
- 我们可以通过 `START TRANSACTION READ WRITE` 语句开启一个读写事务，或者使用 `BEGIN` 、 `START TRANSACTION` 语句开启的事务默认也算是读写事务。

  在读写事务中可以对表执行增删改查操作。

如果某个事务执行过程中对某个表执行了增、删、改操作，那么 `InnoDB` 存储引擎就会给它分配一个独一无二的 `事务id` ，分配方式如下：

- 对于只读事务来说，只有在它第一次对某个用户创建的临时表执行增、删、改操作时才会为这个事务分配一个 `事务id` ，否则的话是不分配 `事务id` 的。

  > 小贴士：
  > 我们前边说过对某个查询语句执行EXPLAIN分析它的查询计划时，有时候在Extra列会看到Using temporary的提示，这个表明在执行该查询语句时会用到内部临时表。这个所谓的内部临时表和我们手动用CREATE TEMPORARY TABLE创建的用户临时表并不一样，在事务回滚时并不需要把执行SELECT语句过程中用到的内部临时表也回滚，在执行SELECT语句用到内部临时表时并不会为它分配事务id。

- 对于读写事务来说，只有在它第一次对某个表（包括用户创建的临时表）执行增、删、改操作时才会为这个事务分配一个 `事务id` ，否则的话也是不分配 `事务id` 的。

  有的时候虽然我们开启了一个读写事务，但是在这个事务中全是查询语句，并没有执行增、删、改的语句，那也就意味着这个事务并不会被分配一个 `事务id` 。

说了半天， `事务id` 有啥子用？这个先保密哈，后边会一步步的详细唠叨。现在只要知道只有在事务对表中的记录做改动时才会为这个事务分配一个唯一的 `事务id` 。

> 小贴士：
> 上边描述的事务id分配策略是针对MySQL 5.7来说的，前边的版本的分配方式可能不同～

22.2.2 事务id是怎么生成的

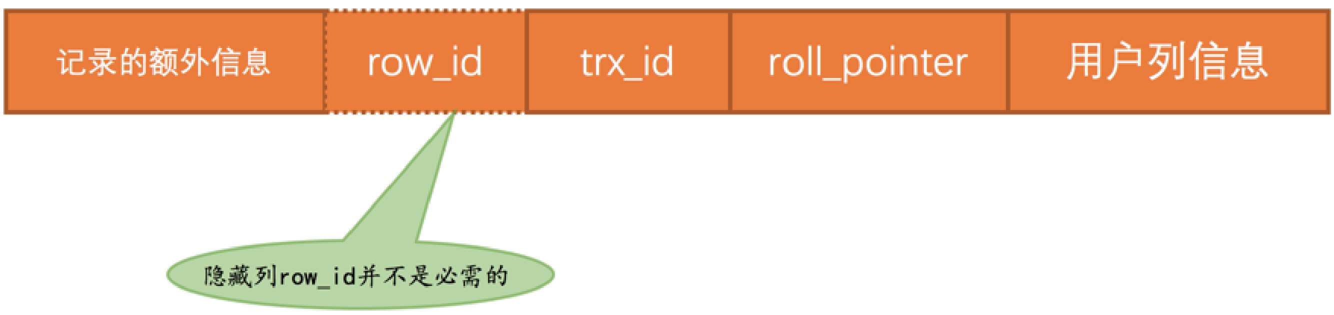
这个 事务id 本质上就是一个数字，它的分配策略和我们前边提到的对隐藏列 row_id （当用户没有为表创建主键和 UNIQUE 键时 InnoDB 自动创建的列）的分配策略大抵相同，具体策略如下：

- 服务器会在内存中维护一个全局变量，每当需要为某个事务分配一个 事务id 时，就会把该变量的值当作 事务id 分配给该事务，并且把该变量自增1。
- 每当这个变量的值为 256 的倍数时，就会将该变量的值刷新到系统表空间的页号为 5 的页面中一个称之为 Max Trx ID 的属性处，这个属性占用 8 个字节的存储空间。
- 当系统下一次重新启动时，会将上边提到的 Max Trx ID 属性加载到内存中，将该值加上256之后赋值给我们前边提到的全局变量（因为在上次关机时该全局变量的值可能大于 Max Trx ID 属性值）。

这样就可以保证整个系统中分配的 事务id 值是一个递增的数字。先被分配 id 的事务得到的是较小的 事务id ，后被分配 id 的事务得到的是较大的 事务id 。

22.2.3 trx_id隐藏列

我们前边唠叨 InnoDB 记录行格式的时候重点强调过：聚簇索引的记录除了会保存完整的用户数据以外，而且还会自动添加名为trx_id、roll_pointer的隐藏列，如果用户没有在表中定义主键以及UNIQUE键，还会自动添加一个名为row_id的隐藏列。所以一条记录在页面中的真实结构看起来就是这样的：



其中的 trx_id 列其实还蛮好理解的，就是某个对这个聚簇索引记录做改动的语句所在的事务对应的 事务id 而已（此处的改动可以是 INSERT 、 DELETE 、 UPDATE 操作）。至于 roll_pointer 隐藏列我们后边分析~

22.3 undo日志的格式

为了实现事务的 原子性 ， InnoDB 存储引擎在实际进行增、删、改一条记录时，都需要先把对应的 undo日志 记下来。一般每对一条记录做一次改动，就对应着一条 undo日志 ，但在某些更新记录的操作中，也可能会对应着2条 undo日志 ，这个我们后边会仔细唠叨。一个事务在执行过程中可能新增、删除、更新若干条记录，也就是说需要记录很多条对应的 undo日志 ，这些 undo日志 会被从 0 开始编号，也就是说根据生成的顺序分别被称为 第0号undo日志 、 第1号undo日志 、...、 第n号undo日志 等，这个编号也被称之为 undo no 。

这些 undo日志 是被记录到类型为 FIL_PAGE_UNDO_LOG （对应的十六进制是 0x0002 ，忘记了页面类型是个啥的同学需要回过头再看看前边的章节）的页面中。这些页面可以从系统表空间中分配，也可以从一种专门存放 undo日志 的表空间，也就是所谓的 undo tablespace 中分配。不过关于如何分配存储 undo日志 的页面这个事情我们稍后再说，现在先来看看不同操作都会产生什么样子的 undo日志 吧~ 为了故事的顺利发展，我们先来创建一个名为 undo_demo 的表：

```
CREATE TABLE undo_demo (
    id INT NOT NULL,
    key1 VARCHAR(100),
    col VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1)
)Engine=InnoDB CHARSET=utf8;
```

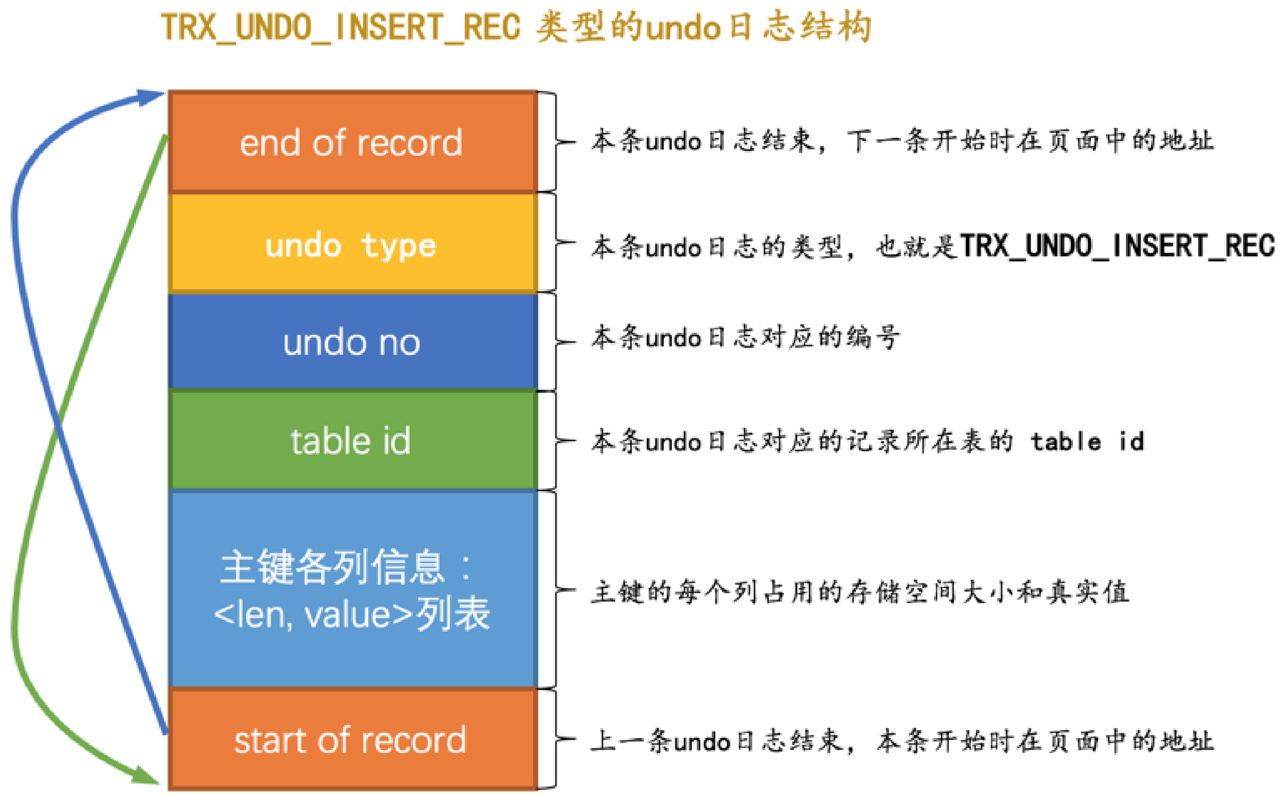
这个表中有3个列，其中 id 列是主键，我们为 key1 列建立了一个二级索引， col 列是一个普通的列。我们前边介绍 InnoDB 的数据字典时说过，每个表都会被分配一个唯一的 table id ，我们可以通过系统数据库 information_schema 中的 innodb_sys_tables 表来查看某个表对应的 table id 是什么，现在我们查看一下 undo_demo 对应的 table id 是多少：

```
mysql> SELECT * FROM information_schema.innodb_sys_tables WHERE name = 'xiaohaizi/undo_dem
o';
+----------+---------------------+------+--------+-------+-------------+------------+-----
----------+------------+
| TABLE_ID | NAME                | FLAG | N_COLS | SPACE | FILE_FORMAT | ROW_FORMAT | ZIP_
PAGE_SIZE | SPACE_TYPE |
+----------+---------------------+------+--------+-------+-------------+------------+-----
----------+------------+
|      138 | xiaohaizi/undo_demo |   33 |      6 |   482 | Barracuda   | Dynamic    |
0 | Single     |
+----------+---------------------+------+--------+-------+-------------+------------+-----
----------+------------+
1 row in set (0.01 sec)
```

从查询结果可以看出， undo_demo 表对应的 table id 为 138 ，先把这个值记住，我们后边有用。

22.3.1 INSERT操作对应的undo日志

我们前边说过，当我们向表中插入一条记录时会有 乐观插入 和 悲观插入 的区分，但是不管怎么插入，最终导致的结果就是这条记录被放到了一个数据页中。如果希望回滚这个插入操作，那么把这条记录删除就好了，也就是说在写对应的 undo 日志时，主要是把这条记录的主键信息记上。所以设计 InnoDB 的大叔设计了一个类型为 TRX_UNDO_INSERT_REC 的 undo日志 ，它的完整结构如下图所示：



根据示意图我们强调几点：

- undo no 在一个事务中是从 0 开始递增的，也就是说只要事务没提交，每生成一条 undo日志 ，那么该条日志的 undo no 就增1。
- 如果记录中的主键只包含一个列，那么在类型为 TRX_UNDO_INSERT_REC 的 undo日志 中只需要把该列占用的存储空间大小和真实值记录下来，如果记录中的主键包含多个列，那么每个列占用的存储空间大小和对应的真实值都需要记录下来（图中的 len 就代表列占用的存储空间大小， value 就代表列的真实值）。

> 小贴士：
> 当我们向某个表中插入一条记录时，实际上需要向聚簇索引和所有的二级索引都插入一条记录。不过记录undo日志时，我们只需要考虑向聚簇索引插入记录时的情况就好了，因为其实聚簇索引记录和二级索引记录是一一对应的，我们在回滚插入操作时，只需要知道这条记录的主键信息，然后根据主键信息做对应的删除操作，做删除操作时就会顺带着把所有二级索引中相应的记录也删除掉。后边说到的DELETE操作和UPDATE操作对应的undo日志也都是针对聚簇索引记录而言的，我们之后就不强调了。

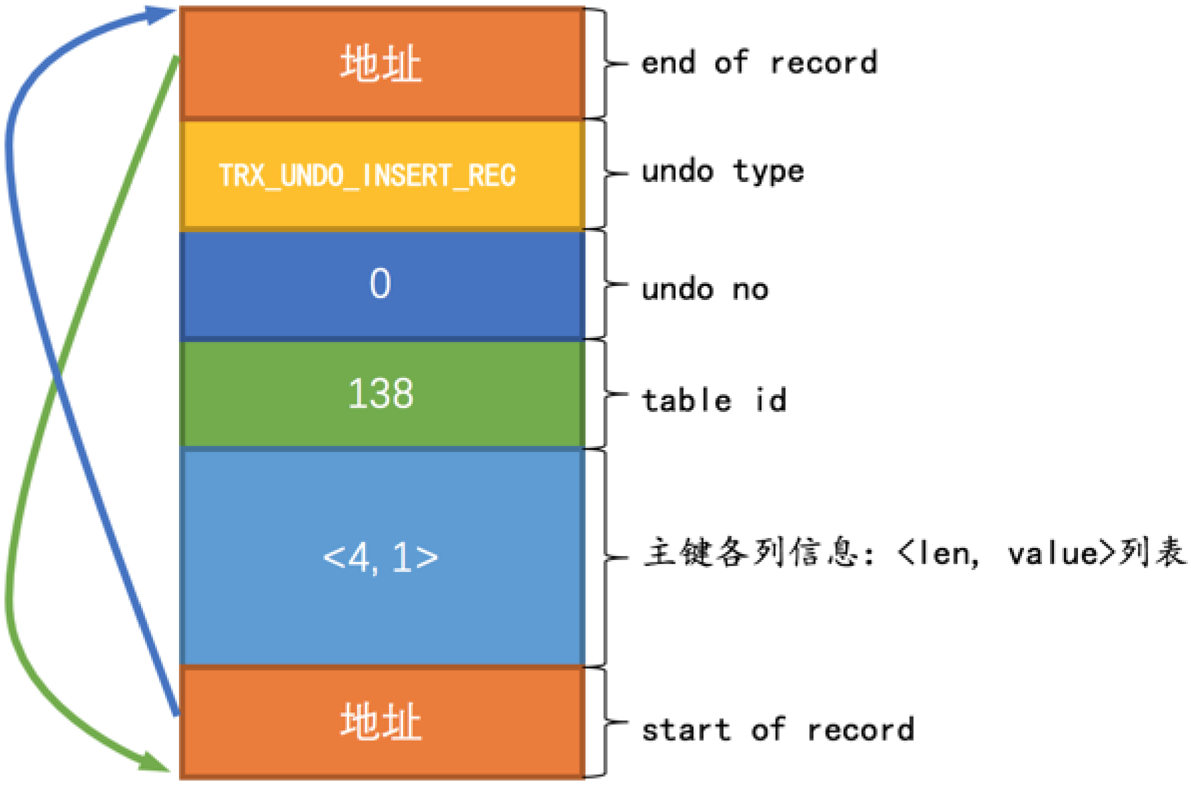
现在我们向 undo_demo 中插入两条记录：

```
BEGIN;  # 显式开启一个事务，假设该事务的id为100

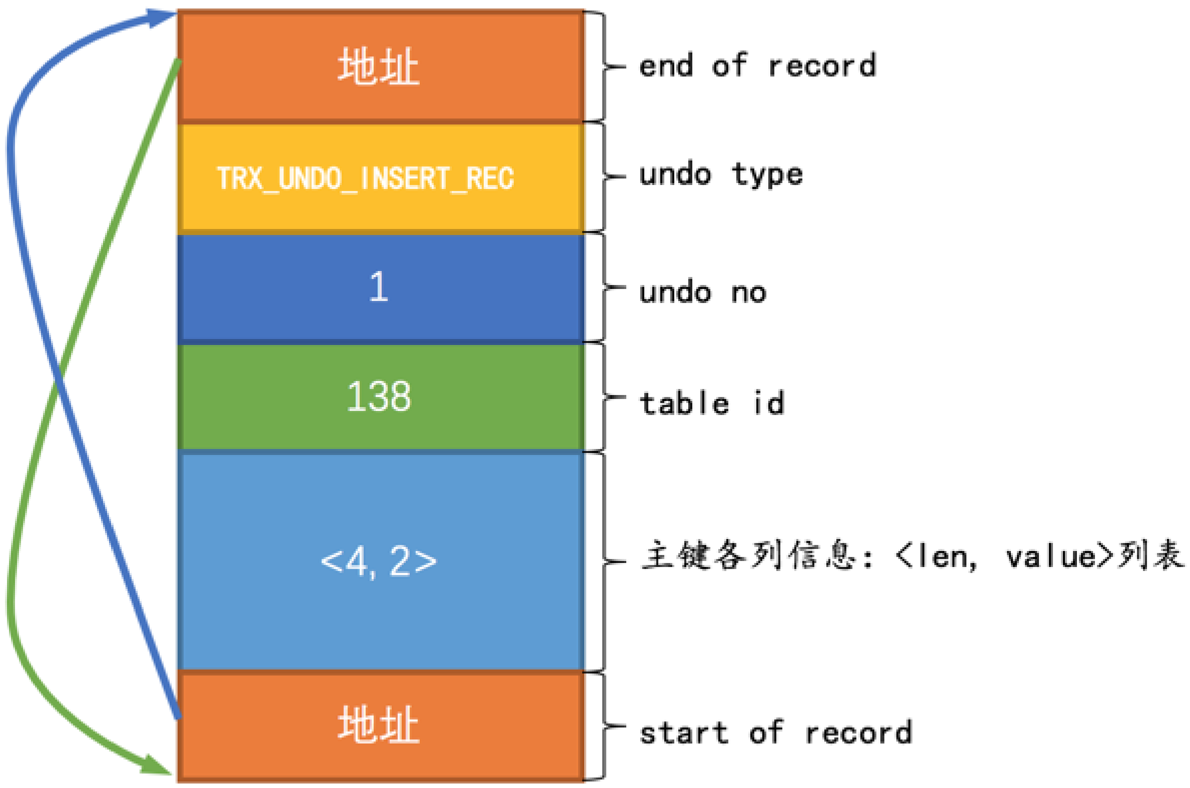
# 插入两条记录
INSERT INTO undo_demo(id, key1, col)
    VALUES (1, 'AWM', '狙击枪'), (2, 'M416', '步枪');
```

因为记录的主键只包含一个 id 列，所以我们在对应的 undo日志 中只需要将待插入记录的 id 列占用的存储空间长度（ id 列的类型为 INT ， INT 类型占用的存储空间长度为 4 个字节）和真实值记录下来。本例中插入了两条记录，所以会产生两条类型为 TRX_UNDO_INSERT_REC 的 undo日志 ：

- 第一条 undo日志 的 undo no 为 0 ，记录主键占用的存储空间长度为 4 ，真实值为 1 。画一个示意图就是这样：



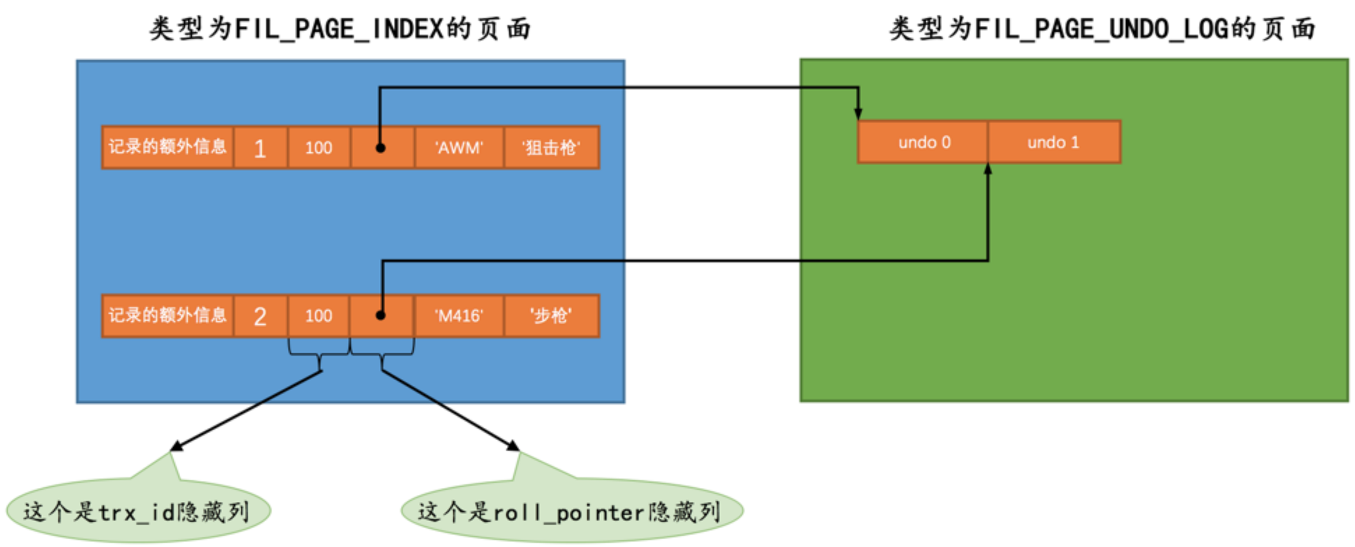
- 第二条 undo日志 的 undo no 为 1 ，记录主键占用的存储空间长度为 4 ，真实值为 2 。画一个示意图就是这样（与第一条 undo日志 对比， undo no 和主键各列信息有不同）：



小贴士：
为了最大限度的节省undo日志占用的存储空间，和我们前边说过的redo日志类似，设计InnoDB的大叔会给undo日志中的某些属性进行压缩处理，具体的压缩细节我们就不唠叨了。

22.3.1.1 roll pointer隐藏列的含义

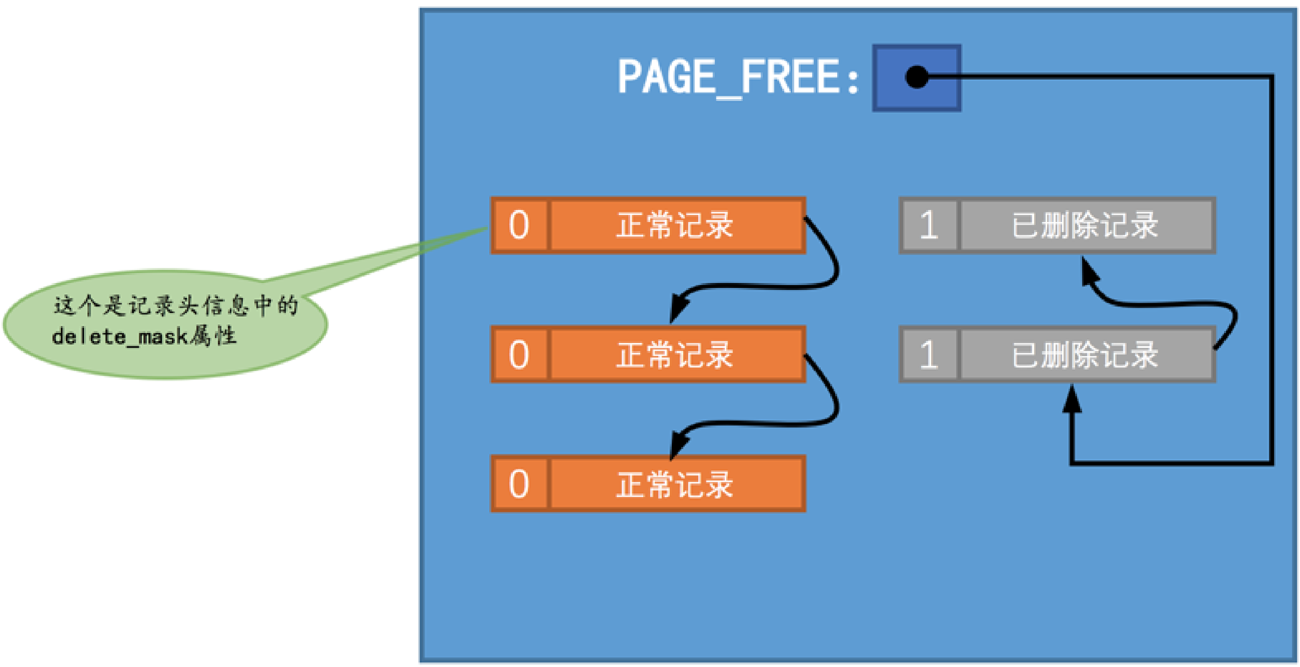
是时候揭开 roll_pointer 的真实面纱了，这个占用 7 个字节的字段其实一点都不神秘，本质上就是一个指向记录对应的 undo日志 的一个指针。比方说我们上边向 undo_demo 表里插入了2条记录，每条记录都有与其对应的一条 undo日志 。记录被存储到了类型为 FIL_PAGE_INDEX 的页面中（就是我们前边一直所说的 数据页 ）， undo日志 被存放到了类型为 FIL_PAGE_UNDO_LOG 的页面中。效果如图所示：



从图中也可以更直观的看出来， roll_pointer 本质就是一个指针，指向记录对应的undo日志。不过这 7 个字节的 roll_pointer 的每一个字节具体的含义我们后边唠叨完如何分配存储 undo 日志的页面之后再具体说哈～

22.3.2 DELETE操作对应的undo日志

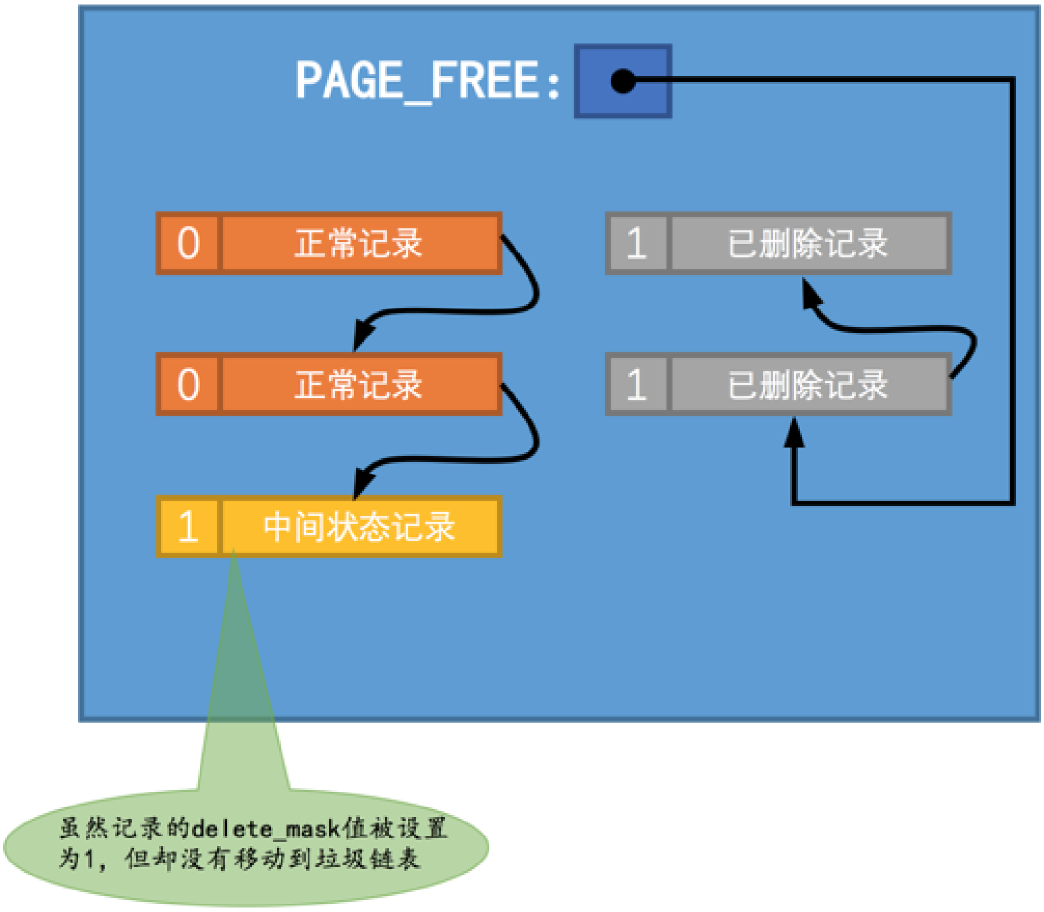
我们知道插入到页面中的记录会根据记录头信息中的 next_record 属性组成一个单向链表，我们把这个链表称之为 正常记录链表 ；我们在前边唠叨数据页结构的时候说过，被删除的记录其实也会根据记录头信息中的 next_record 属性组成一个链表，只不过这个链表中的记录占用的存储空间可以被重新利用，所以也称这个链表为 垃圾链表 。 Page Header 部分有一个称之为 PAGE_FREE 的属性，它指向由被删除记录组成的垃圾链表中的头节点。为了故事的顺利发展，我们先画一个图，假设此刻某个页面中的记录分布情况是这样的（这个不是 undo_demo 表中的记录，只是我们随便举的一个例子）：



为了突出主题，在这个简化版的示意图中，我们只把记录的 delete_mask 标志位展示了出来。从图中可以看出， 正常记录链表 中包含了3条正常记录， 垃圾链表 里包含了2条已删除记录，在 垃圾链表 中的这些记录占用的存储空间可以被重新利用。页面的 Page Header 部分的 PAGE_FREE 属性的值代表指向 垃圾链表 头节点的指针。假设现在我们准备使用 DELETE 语句把 正常记录链表 中的最后一条记录给删除掉，其实这个删除的过程需要经历两个阶段：

- 阶段一：仅仅将记录的 delete_mask 标识位设置为 1 ，其他的不做修改（其实会修改记录的 trx_id 、 roll_pointer 这些隐藏列的值）。设计 InnoDB 的大叔把这个阶段称之为 delete mark 。

  把这个过程画下来就是这样：

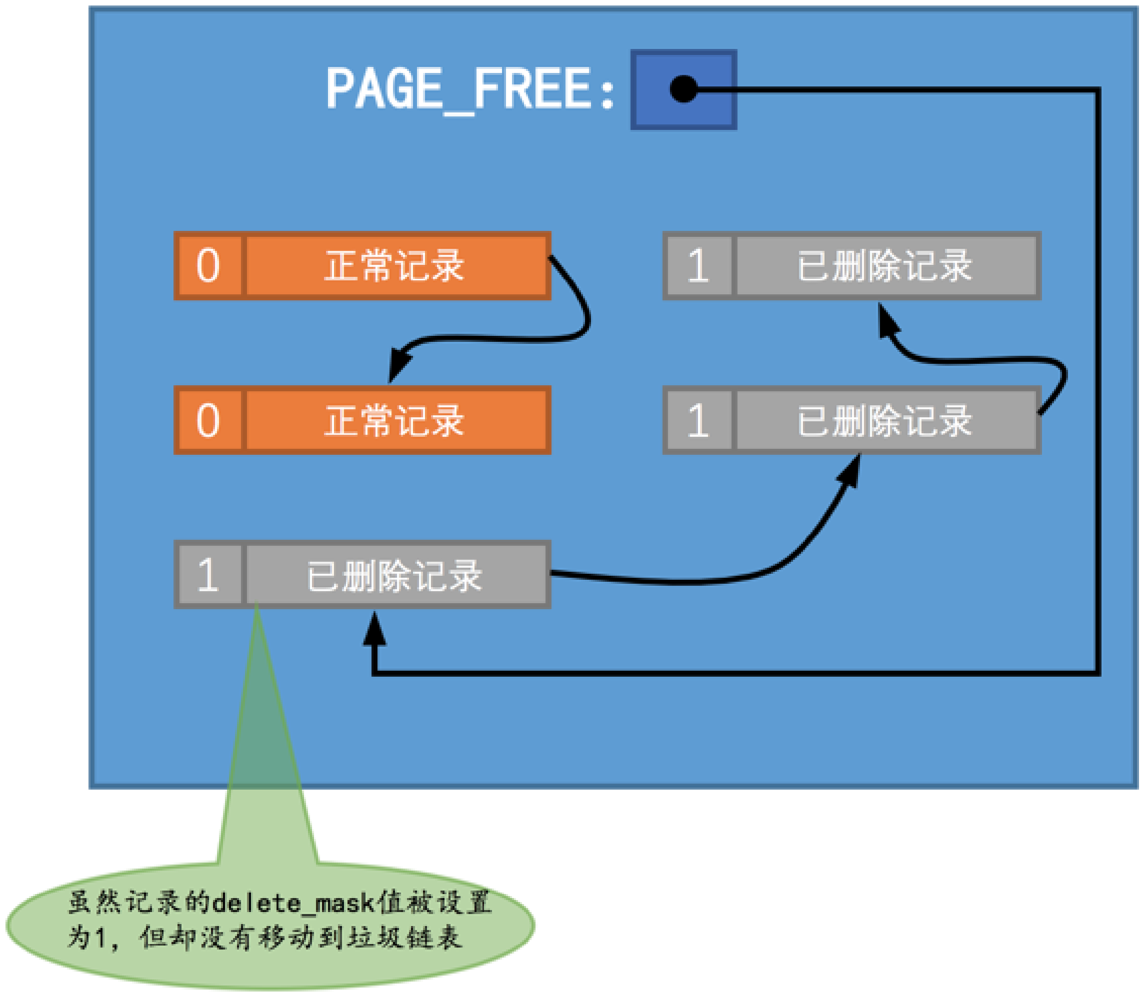
  

  可以看到， 正常记录链表 中的最后一条记录的 delete_mask 值被设置为 1 ，但是并没有被加入到 垃圾链表 。也就是此时记录处于一个 中间状态 ，跟猪八戒照镜子——里外不是人似的。在删除语句所在的事务提交之前，被删除的记录一直都处于这种所谓的 中间状态 。

  > 小贴士：
  > 为啥会有这种奇怪的中间状态呢？其实主要是为了实现一个称之为MVCC的功能，哈哈，稍后再介绍。

- 阶段二：当该删除语句所在的事务提交之后，会有专门的线程后来真正的把记录删除掉。所谓真正的删除就是把该记录从 正常记录链表 中移除，并且加入到 垃圾链表 中，然后还要调整一些页面的其他信息，比如页面中的用户记录数量 PAGE_N_RECS 、上次插入记录的位置 PAGE_LAST_INSERT 、垃圾链表头节点的指针 PAGE_FREE 、页面中可重用的字节数量 PAGE_GARBAGE 、还有页目录的一些信息等等。设计 InnoDB 的大叔把这个阶段称之为 purge 。

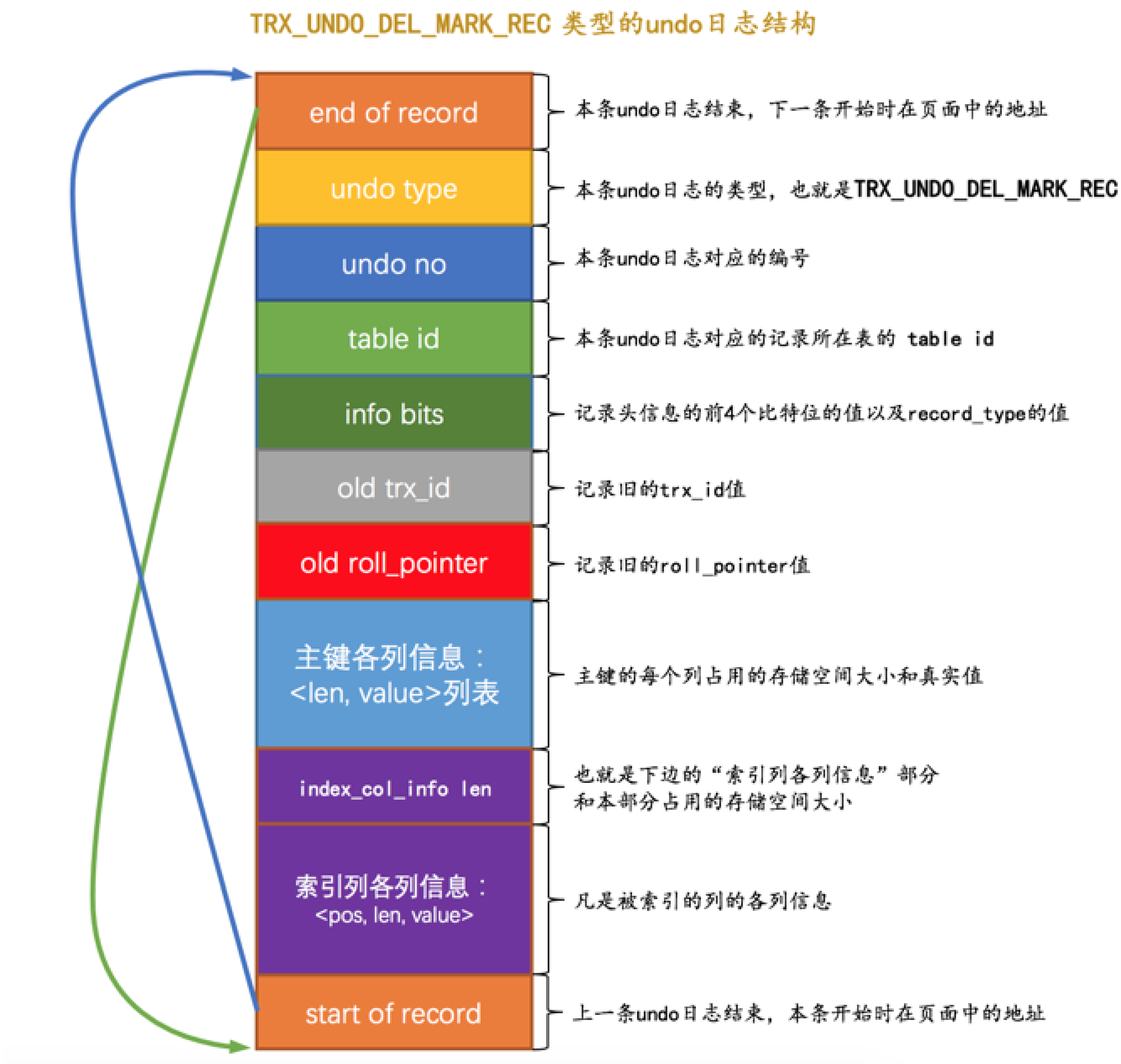
  把 阶段二 执行完了，这条记录就算是真正的被删除掉了。这条已删除记录占用的存储空间也可以被重新利用了。画下来就是这样：



对照着图我们还要注意一点，将被删除记录加入到 垃圾链表 时，实际上加入到链表的头节点处，会跟着修改 PAGE_FREE 属性的值。

小贴士：
页面的Page Header部分有一个PAGE_GARBAGE属性，该属性记录着当前页面中可重用存储空间占用的总字节数。每当有已删除记录被加入到垃圾链表后，都会把这个PAGE_GARBAGE属性的值加上该已删除记录占用的存储空间大小。PAGE_FREE指向垃圾链表的头节点，之后每当新插入记录时，首先判断PAGE_FREE指向的头节点代表的已删除记录占用的存储空间是否足够容纳这条新插入的记录，如果不可以容纳，就直接向页面中申请新的空间来存储这条记录（是的，你没看错，并不会尝试遍历整个垃圾链表，找到一个可以容纳新记录的节点）。如果可以容纳，那么直接重用这条已删除记录的存储空间，并且把PAGE_FREE指向垃圾链表中的下一条已删除记录。但是这里有一个问题，如果新插入的那条记录占用的存储空间大小小于垃圾链表的头节点占用的存储空间大小，那就意味头节点对应的记录占用的存储空间里有一部分空间用不到，这部分空间就被称之为碎片空间。那这些碎片空间岂不是永远都用不到了么？其实也不是，这些碎片空间占用的存储空间大小会被统计到PAGE_GARBAGE属性中，这些碎片空间在整个页面快使用完前并不会被重新利用，不过当页面快满时，如果再插入一条记录，此时页面中并不能分配一条完整记录的空间，这时候会首先看一看PAGE_GARBAGE的空间和剩余可利用的空间加起来是不是可以容纳下这条记录，如果可以的话，InnoDB会尝试重新组织页内的记录，重新组织的过程就是先开辟一个临时页面，把页面内的记录依次插入一遍，因为依次插入时并不会产生碎片，之后再把临时页面的内容复制到本页面，这样就可以把那些碎片空间都解放出来（很显然重新组织页面内的记录比较耗费性能）。

从上边的描述中我们也可以看出来，在删除语句所在的事务提交之前，只会经历 阶段一 ，也就是 delete mark 阶段（提交之后我们就不用回滚了，所以只需考虑对删除操作的 阶段一 做的影响进行回滚）。设计 InnoDB 的大叔为此设计了一种称之为 TRX_UNDO_DEL_MARK_REC 类型的 undo日志 ，它的完整结构如下图所示：



额滴个神呐，这个里边的属性也太多了点儿吧～ （其实大部分属性的意思我们上边已经介绍过了） 是的，的确有点多，不过大家千万不要在意，如果记不住千万不要勉强自己，我这里把它们都列出来让大家混个脸熟而已。劳烦大家先克服一下密集恐急症，再抬头大致看一遍上边的这个类型为 TRX_UNDO_DEL_MARK_REC 的 undo日志 中的属性，特别注意一下这几点：

- 在对一条记录进行 delete mark 操作前，需要把该记录的旧的 trx_id 和 roll_pointer 隐藏列的值都给记到对应的 undo日志 中来，就是我们图中显示的 old trx_id 和 old roll_pointer 属性。这样有一个好处，那就是可以通过 undo日志 的 old roll_pointer 找到记录在修改之前对应的 undo 日志。比方说在一个事务中，我们先插入了一条记录，然后又执行对该记录的删除操作，这个过程的示意图就是这样：



从图中可以看出来，执行完 delete mark 操作后，它对应的 undo 日志和 INSERT 操作对应的 undo 日志就串成了一个链表。这个很有意思啊，这个链表就称之为 版本链 ，现在貌似看不出这个 版本链 有啥用，等我们再往后看看，讲完 UPDATE 操作对应的 undo 日志后，这个所谓的 版本链 就慢慢的展现出它的牛逼之处了。

- 与类型为 TRX_UNDO_INSERT_REC 的 undo日志 不同，类型为 TRX_UNDO_DEL_MARK_REC 的 undo 日志还多了一个 索引列各列信息 的内容，也就是说如果某个列被包含在某个索引中，那么它的相关信息就应该被记录到这个 索引列各列信息 部分，所谓的相关信息包括该列在记录中的位置（用 pos 表示），该列占用的存储空间大小（用 len 表示），该列实际值（用 value 表示）。所以 索引列各列信息 存储的内容实质上就是 <pos, len, value> 的一个列表。这部分信息主要是用在事务提交后，对该 中间状态记录 做真正删除的阶段二，也就是 purge 阶段中使用的，具体如何使用现在我们可以忽略~

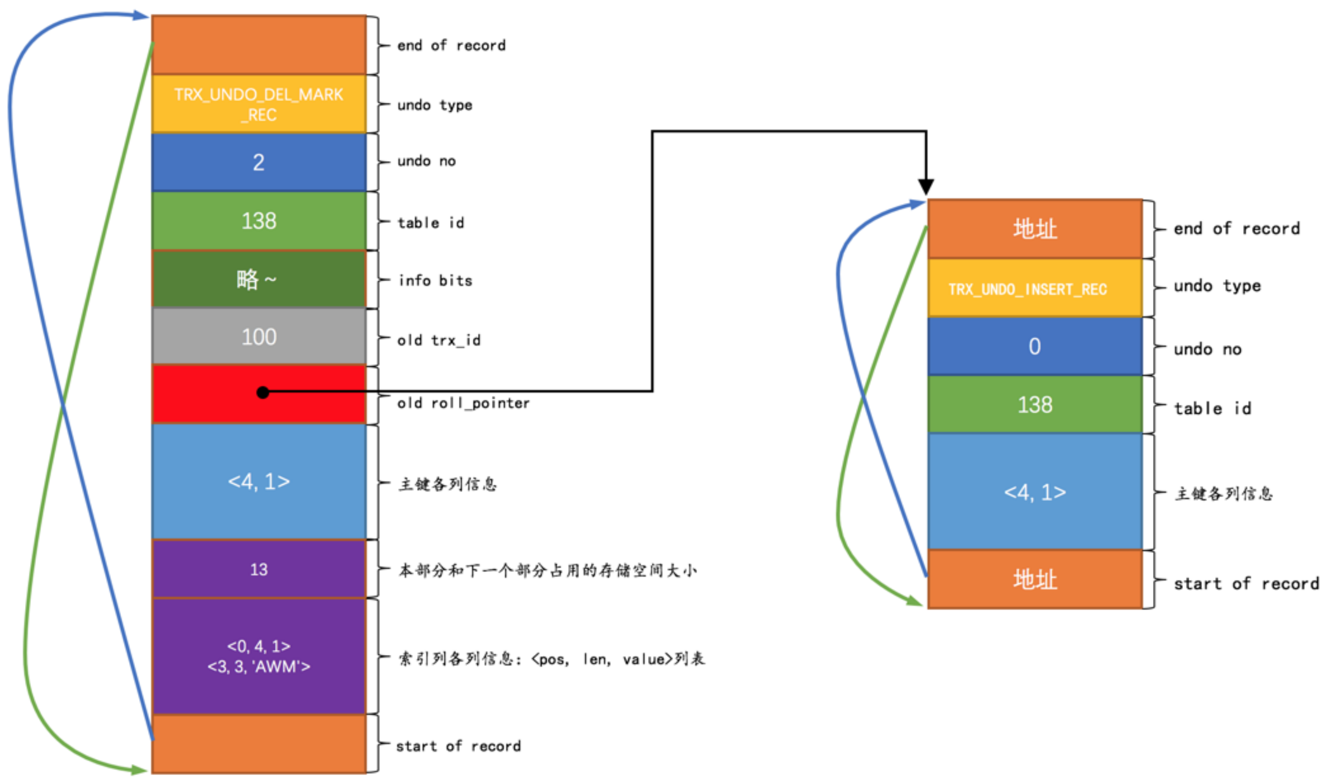
该介绍的我们介绍完了，现在继续在上边那个事务id为 100 的事务中删除一条记录，比如我们把 id 为1的那条记录删除掉：

```
BEGIN;  # 显式开启一个事务，假设该事务的id为100

# 插入两条记录
INSERT INTO undo_demo(id, key1, col)
    VALUES (1, 'AWM', '狙击枪'), (2, 'M416', '步枪');

# 删除一条记录
DELETE FROM undo_demo WHERE id = 1;
```

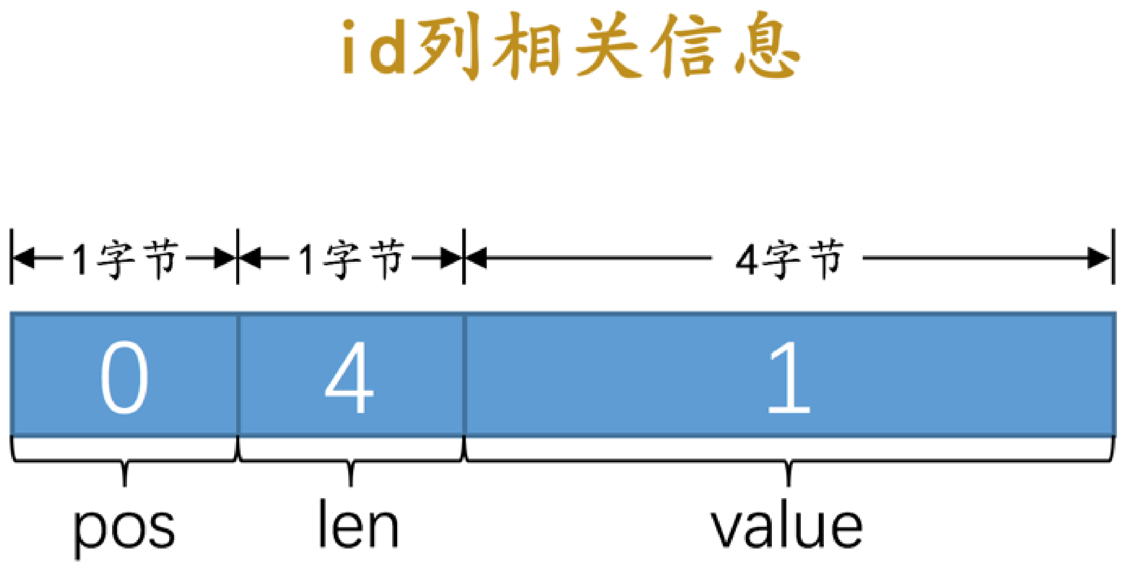
这个 delete mark 操作对应的 undo日志 的结构就是这样：



对照着这个图，我们得注意下边几点：

- 因为这条 undo 日志是 id 为 100 的事务中产生的第3条 undo 日志，所以它对应的 undo no 就是 2 。
- 在对记录做 delete mark 操作时，记录的 trx_id 隐藏列的值是 100 （也就是说对该记录最近的一次修改就发生在本事务中），所以把 100 填入 old trx_id 属性中。然后把记录的 roll_pointer 隐藏列的值取出来，填入 old roll_pointer 属性中，这样就可以通过 old roll_pointer 属性值找到最近一次对该记录做改动时产生的 undo日志 。
- 由于 undo_demo 表中有2个索引：一个是聚簇索引，一个是二级索引 idx_key1 。只要是包含在索引中的列，那么这个列在记录中的位置（ pos ），占用存储空间大小（ len ）和实际值（ value ）就需要存储到 undo日志 中。
  - 对于主键来说，只包含一个 id 列，存储到 undo日志 中的相关信息分别是：
    - pos ： id 列是主键，也就是在记录的第一个列，它对应的 pos 值为 0 。 pos 占用1个字节来存储。
    - len ： id 列的类型为 INT ，占用4个字节，所以 len 的值为 4 。 len 占用1个字节来存储。
    - value ：在被删除的记录中 id 列的值为 1 ，也就是 value 的值为 1 。 value 占用4个字节来存储。

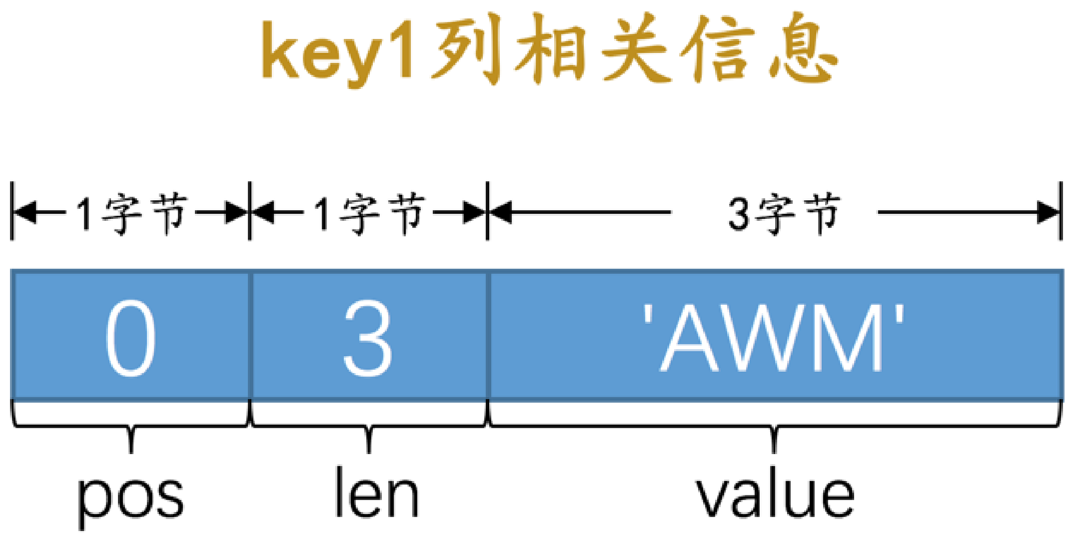
    画一个图演示一下就是这样：



所以对于 id 列来说，最终存储的结果就是 <0, 4, 1> ，存储这些信息占用的存储空间大小为 1 + 1 + 4 = 6 个字节。

- 对于 idx_key1 来说，只包含一个 key1 列，存储到 undo日志 中的相关信息分别是：
  - pos ： key1 列是排在 id 列、 trx_id 列、 roll_pointer 列之后的，它对应的 pos 值为 3 。 pos 占用1个字节来存储。
  - len ： key1 列的类型为 VARCHAR(100) ，使用 utf8 字符集，被删除的记录实际存储的内容是 AWM ，所以一共占用3个字节，也就是所以 len 的值为 3 。 len 占用1个字节来存储。
  - value ：在被删除的记录中 key1 列的值为 AWM ，也就是 value 的值为 AWM 。 value 占用3个字节来存储。

画一个图演示一下就是这样：



所以对于 key1 列来说，最终存储的结果就是 <3, 3, 'AWM'> ，存储这些信息占用的存储空间大小为 1 + 1 + 3 = 5 个字节。

从上边的叙述中可以看到， <0, 4, 1> 和 <3, 3, 'AWM'> 共占用 11 个字节。然后 index_col_info len 本身占用 2 个字节，所以加起来一共占用 13 个字节，把数字 13 就填到了 index_col_info len 的属性中。

22.3.3 UPDATE操作对应的undo日志

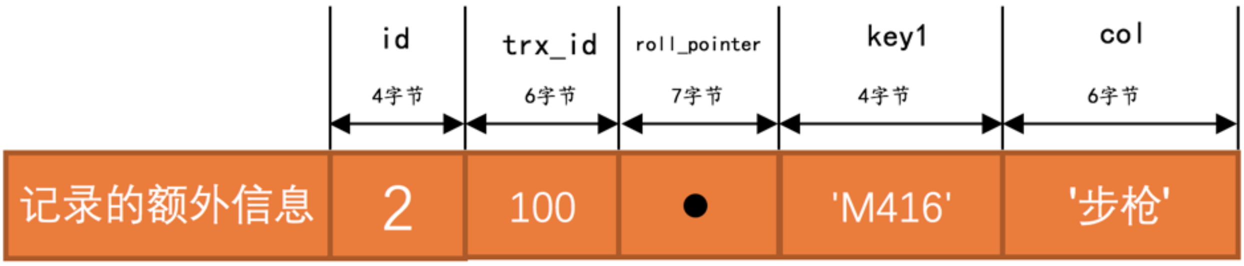
在执行 UPDATE 语句时， InnoDB 对更新主键和不更新主键这两种情况有截然不同的处理方案。

22.3.3.1 不更新主键的情况

在不更新主键的情况下，又可以细分为被更新的列占用的存储空间不发生变化和发生变化的情况。

- 就地更新（in-place update）

  更新记录时，对于被更新的每个列来说，如果更新后的列和更新前的列占用的存储空间都一样大，那么就可以进行 就地更新 ，也就是直接在原记录的基础上修改对应列的值。再次强调一边，是每个列在更新前后占用的存储空间一样大，有任何一个被更新的列更新前比更新后占用的存储空间大，或者更新前比更新后占用的存储空间小都不能进行 就地更新 。比方说现在 undo_demo 表里还有一条 id 值为 2 的记录，它的各个列占用的大小如图所示（因为采用 utf8 字符集，所以 '步枪' 这两个字符占用6个字节）：



  假如我们有这样的 UPDATE 语句：

```
UPDATE undo_demo
    SET key1 = 'P92', col = '手枪'
    WHERE id = 2;
```

  在这个 UPDATE 语句中， col 列从 步枪 被更新为 手枪 ，前后都占用6个字节，也就是占用的存储空间大小未改变； key1 列从 M416 被更新为 P92 ，也就是从 4 个字节被更新为 3 个字节，这就不满足 就地更新 需要的条件了，所以不能进行 就地更新 。但是如果 UPDATE 语句长这样：

```
UPDATE undo_demo
    SET key1 = 'M249', col = '机枪'
    WHERE id = 2;
```

  由于各个被更新的列在更新前后占用的存储空间是一样大的，所以这样的语句可以执行 就地更新 。

- 先删除掉旧记录，再插入新记录

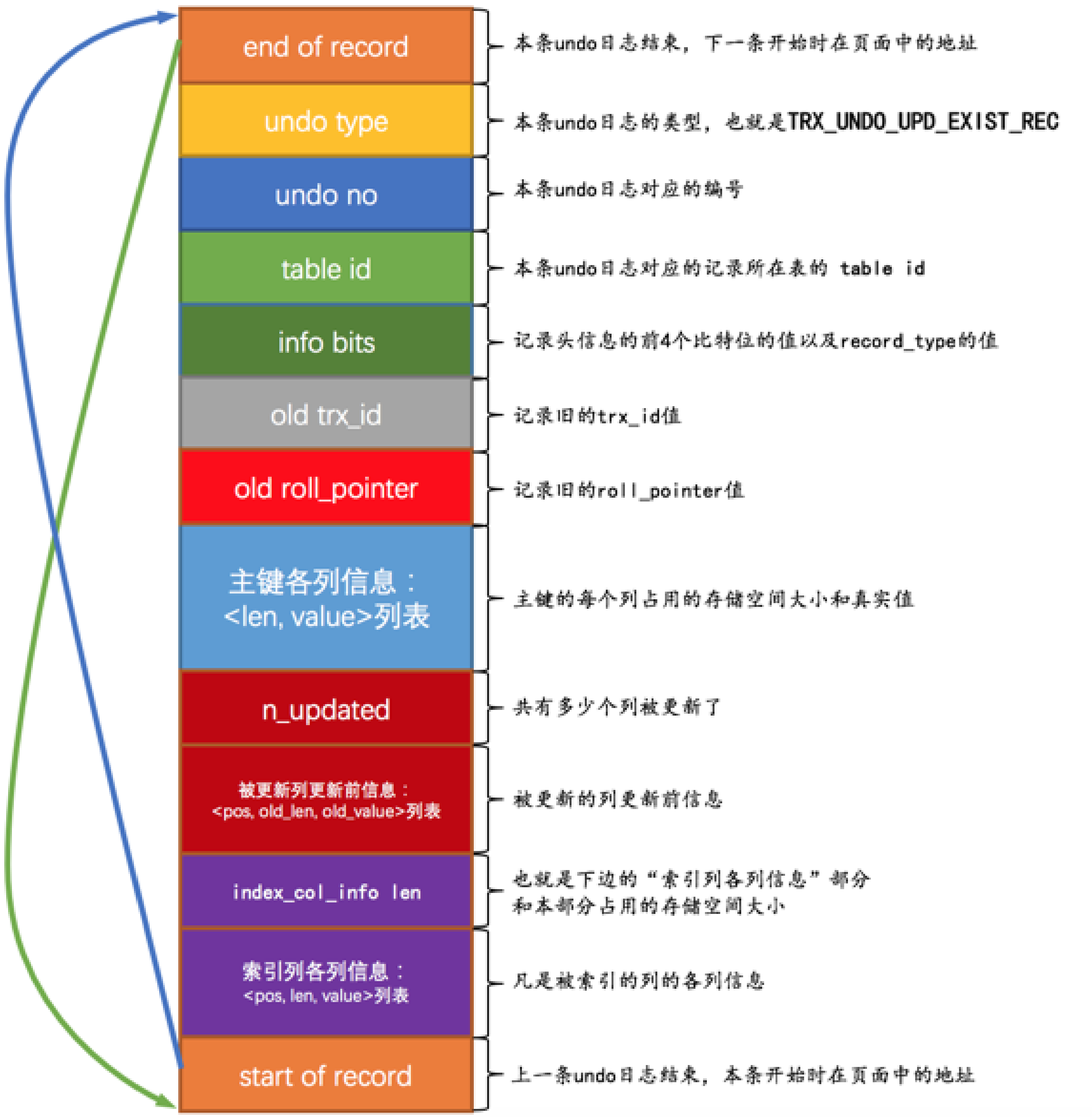
  在不更新主键的情况下，如果有任何一个被更新的列更新前和更新后占用的存储空间大小不一致，那么就需要先把这条旧的记录从聚簇索引页面中删除掉，然后再根据更新后列的值创建一条新的记录插入到页面中。

  请注意一下，我们这里所说的 删除 并不是 delete mark 操作，而是真正的删除掉，也就是把这条记录从 正常记录链表 中移除并加入到 垃圾链表 中，并且修改页面中相应的统计信息（比如 PAGE_FREE 、 PAGE_GARBAGE 等这些信息）。不过这里做真正删除操作的线程并不是在唠叨 DELETE 语句中做 purge 操作时使用的另外专门的线程，而是由用户线程同步执行真正的删除操作，真正删除之后紧接着就要根据各个列更新后的值创建的新记录插入。

  这里如果新创建的记录占用的存储空间大小不超过旧记录占用的空间，那么可以直接重用被加入到 垃圾链表 中的旧记录所占用的存储空间，否则的话需要在页面中新申请一段空间以供新记录使用，如果本页面内已经没有可用的空间的话，那就需要进行页面分裂操作，然后再插入新记录。

针对 UPDATE 不更新主键的情况（包括上边所说的就地更新和先删除旧记录再插入新记录），设计 InnoDB 的大叔们设计了一种类型为 TRX_UNDO_UPD_EXIST_REC 的 undo日志，它的完整结构如下：

TRX_UNDO_UPD_EXIST_REC类型的undo日志结构



其实大部分属性和我们介绍过的 TRX_UNDO_DEL_MARK_REC 类型的 undo日志 是类似的，不过还是要注意这么几点：

- n_updated 属性表示本条 UPDATE 语句执行后将有几个列被更新，后边跟着的 <pos, old_len, old_value> 分别表示被更新列在记录中的位置、更新前该列占用的存储空间大小、更新前该列的真实值。
- 如果在 UPDATE 语句中更新的列包含索引列，那么也会添加 索引列各列信息 这个部分，否则的话是不会添加这个部分的。

现在继续在上边那个事务id为100的事务中更新一条记录，比如我们把id为2的那条记录更新一下：

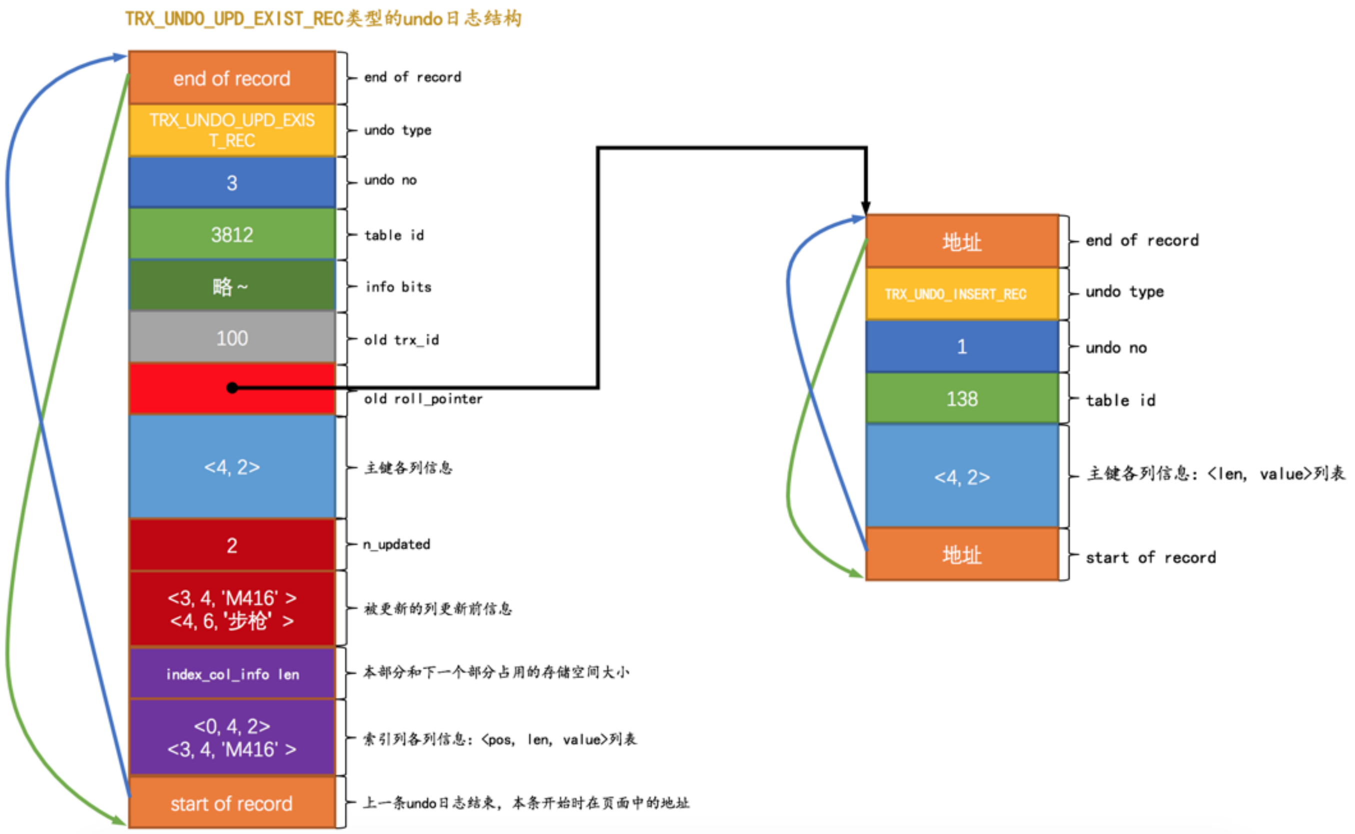
```
BEGIN;  # 显式开启一个事务，假设该事务的id为100

# 插入两条记录
INSERT INTO undo_demo(id, key1, col)
    VALUES (1, 'AWM', '狙击枪'), (2, 'M416', '步枪');

# 删除一条记录
DELETE FROM undo_demo WHERE id = 1;

# 更新一条记录
UPDATE undo_demo
    SET key1 = 'M249', col = '机枪'
    WHERE id = 2;
```

这个 UPDATE 语句更新的列大小都没有改动，所以可以采用 就地更新 的方式来执行，在真正改动页面记录时，会先记录一条类型为 TRX_UNDO_UPD_EXIST_REC 的 undo日志 ，长这样：



对照着这个图我们注意一下这几个地方：

- 因为这条 undo日志 是 id 为 100 的事务中产生的第4条 undo日志 ，所以它对应的 undo no 就是3。
- 这条日志的 roll_pointer 指向 undo no 为 1 的那条日志，也就是插入主键值为 2 的记录时产生的那条 undo日志 ，也就是最近一次对该记录做改动时产生的 undo日志 。
- 由于本条 UPDATE 语句中更新了索引列 key1 的值，所以需要记录一下 索引列各列信息 部分，也就是把主键和 key1 列更新前的信息填入。

22.3.3.2 更新主键的情况

在聚簇索引中，记录是按照主键值的大小连成了一个单向链表的，如果我们更新了某条记录的主键值，意味着这条记录在聚簇索引中的位置将会发生改变，比如你将记录的主键值从1更新为10000，如果还有非常多的记录的主键值分布在 1 ~ 10000 之间的话，那么这两条记录在聚簇索引中就有可能离得非常远，甚至中间隔了好多个页面。针对 UPDATE 语句中更新了记录主键值的这种情况， InnoDB 在聚簇索引中分了两步处理：

- 将旧记录进行 delete mark 操作

  高能注意：这里是delete mark操作！这里是delete mark操作！这里是delete mark操作！也就是说在 UPDATE 语句所在的事务提交前，对旧记录只做一个 delete mark 操作，在事务提交后才由专门的线程做purge操作，把它加入到垃圾链表中。这里一定要和我们上边所说的在不更新记录主键值时，先真正删除旧记录，再插入新记录的方式区分开！

  > 小贴士：
  > 之所以只对旧记录做delete mark操作，是因为别的事务同时也可能访问这条记录，如果把它真正的删除加入到垃圾链表后，别的事务就访问不到了。这个功能就是所谓的MVCC，我们后边的章节中会详细唠叨什么是个MVCC。

- 根据更新后各列的值创建一条新记录，并将其插入到聚簇索引中（需重新定位插入的位置）。

  由于更新后的记录主键值发生了改变，所以需要重新从聚簇索引中定位这条记录所在的位置，然后把它插进去。

针对 UPDATE 语句更新记录主键值的这种情况，在对该记录进行 delete mark 操作前，会记录一条类型为 TRX_UNDO_DEL_MARK_REC 的 undo日志 ；之后插入新记录时，会记录一条类型为 TRX_UNDO_INSERT_REC 的 undo日志 ，也就是说每对一条记录的主键值做改动时，会记录2条 undo日志 。这些日志的格式我们上边都唠叨过了，就不赘述了。

> 小贴士：
> 其实还有一种称为TRX_UNDO_UPD_DEL_REC的undo日志的类型我们没有介绍，主要是想避免引入过多的复杂度，如果大家对这种类型的undo日志的使用感兴趣的话，可以额外查一下别的资料。

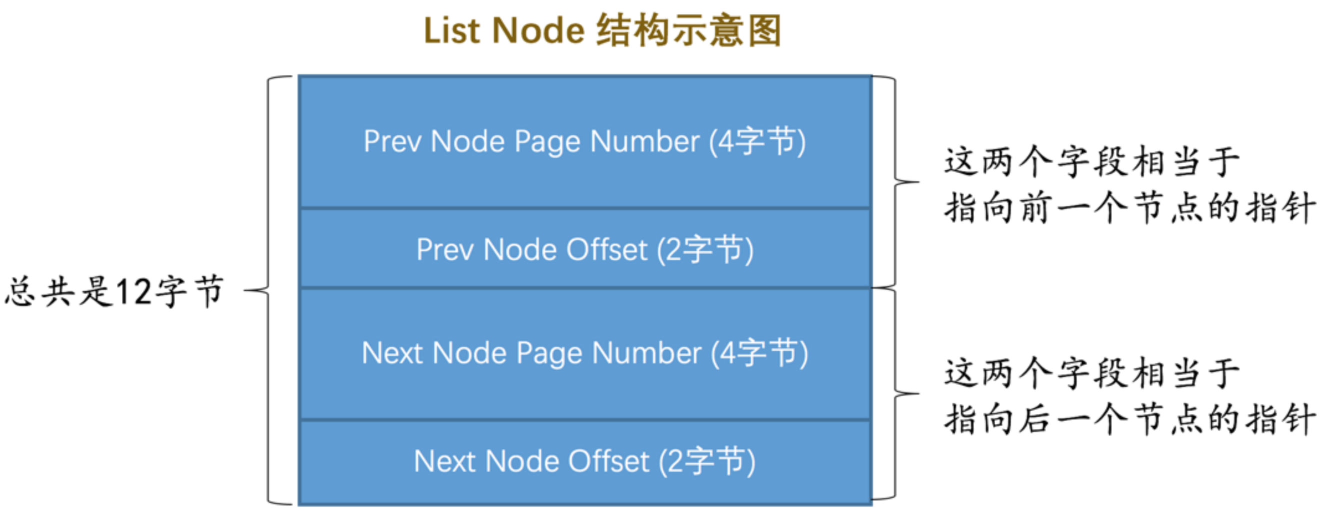
23 第23章 后悔了怎么办-undo日志（下）

标签： MySQL是怎样运行的

上一章我们主要唠叨了为什么需要 undo日志 ，以及 INSERT 、 DELETE 、 UPDATE 这些会对数据做改动的语句都会产生什么类型的 undo日志 ，还有不同类型的 undo日志 的具体格式是什么。本章会继续唠叨这些 undo日志 会被具体写到什么地方，以及在写入过程中需要注意的一些问题。

23.1 通用链表结构

在写入 undo日志 的过程中会使用到多个链表，很多链表都有同样的节点结构，如图所示：

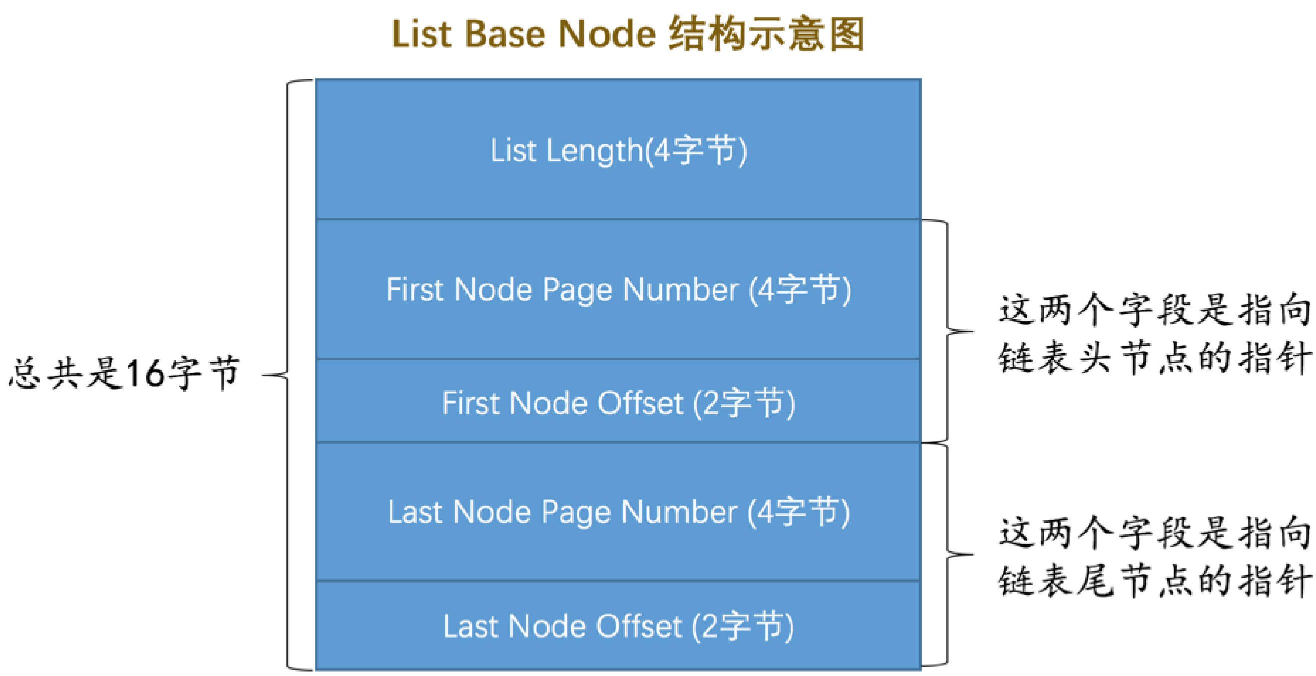


在某个表空间内，我们可以通过一个页的页号和在页内的偏移量来唯一定位一个节点的位置，这两个信息也就相当于指向这个节点的一个指针。所以：

- `Pre Node Page Number` 和 `Pre Node Offset` 的组合就是指向前一个节点的指针
- `Next Node Page Number` 和 `Next Node Offset` 的组合就是指向后一个节点的指针。

整个 `List Node` 占用 `12` 个字节的存储空间。

为了更好的管理链表，设计 `InnoDB` 的大叔还提出了一个基节点的结构，里边存储了这个链表的 `头节点` 、 `尾节点` 以及链表长度信息，基节点的结构示意图如下：

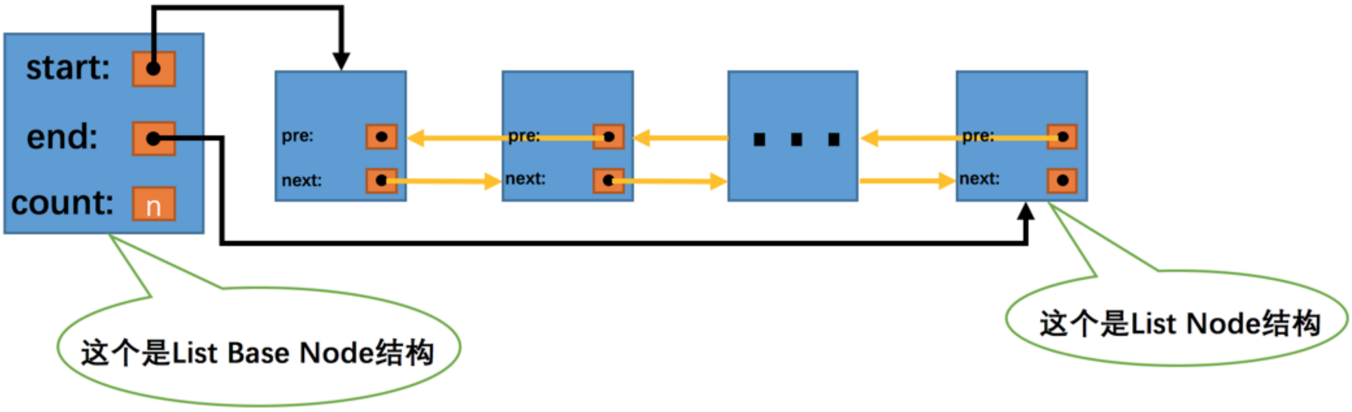


其中：

- `List Length` 表明该链表一共有多少节点。
- `First Node Page Number` 和 `First Node Offset` 的组合就是指向链表头节点的指针。
- `Last Node Page Number` 和 `Last Node Offset` 的组合就是指向链表尾节点的指针。

整个 `List Base Node` 占用 `16` 个字节的存储空间。

所以使用 `List Base Node` 和 `List Node` 这两个结构组成的链表的示意图就是这样：



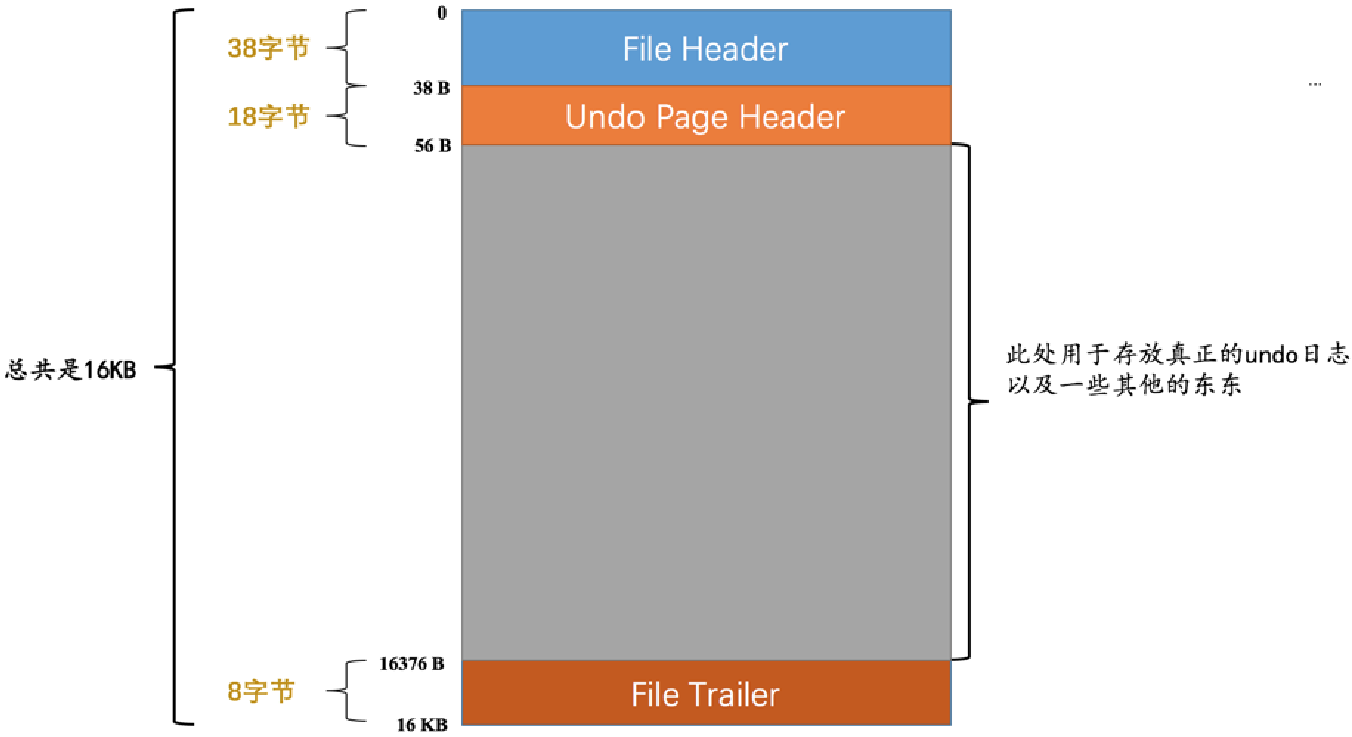
小贴士：

上述链表结构我们在前边的文章中频频提到，尤其是在表空间那一章重点描述过，不过我不敢奢求大家都记住了，所以在这里又强调一遍，希望大家不要嫌我烦，我只是怕大家忘了学习后续内容吃力而已～

23.2 FIL_PAGE_UNDO_LOG页面

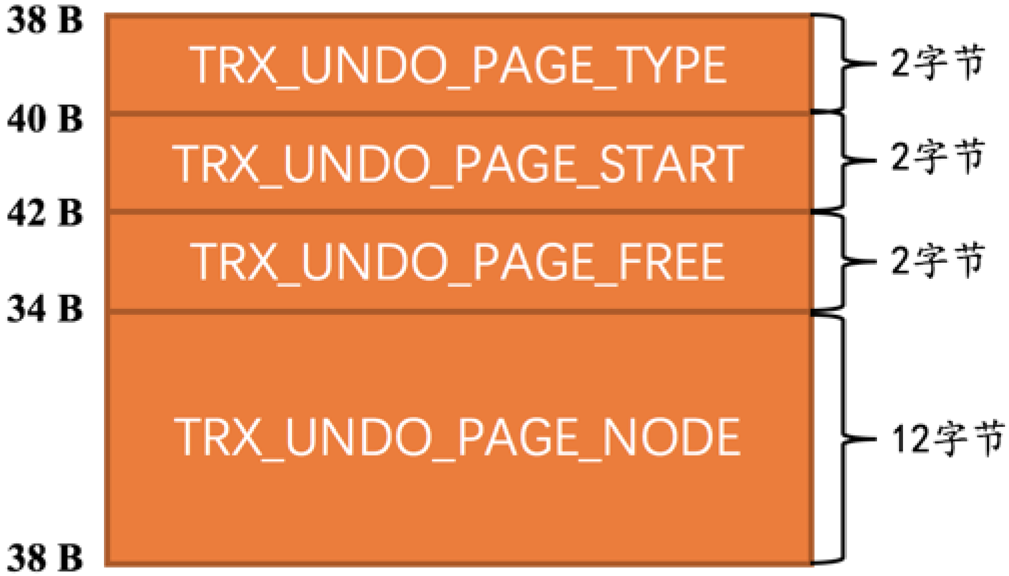
我们前边唠叨表空间的时候说过，表空间其实是由许许多多的页面构成的，页面默认大小为 16KB 。这些页面有不同的类型，比如类型为 FIL_PAGE_INDEX 的页面用于存储聚簇索引以及二级索引，类型为 FIL_PAGE_TYPE_FSP_HDR 的页面用于存储表空间头部信息的，还有其他各种类型的页面，其中有一种称之为 FIL_PAGE_UNDO_LOG 类型的页面是专门用来存储 undo日志 的，这种类型的页面的通用结构如下图所示（以默认的 16KB 大小为例）：

FIL_PAGE_UNDO_LOG页通用结构示意图



“类型为 FIL_PAGE_UNDO_LOG 的页”这种说法太绕口，以后我们就简称为 Undo页面 了哈。上图中的 File Header 和 File Trailer 是各种页面都有的通用结构，我们前边唠叨过很多遍了，这里就不赘述了（忘记了的可以到讲述数据页结构或者表空间的章节中查看）。 Undo Page Header 是 Undo页面 所特有的，我们来看一下它的结构：

Undo Page Header结构示意图



其中各个属性的意思如下：

- TRX_UNDO_PAGE_TYPE ：本页面准备存储什么种类的 undo日志 。

  我们前边介绍了好几种类型的 undo日志 ，它们可以被分为两个大类：

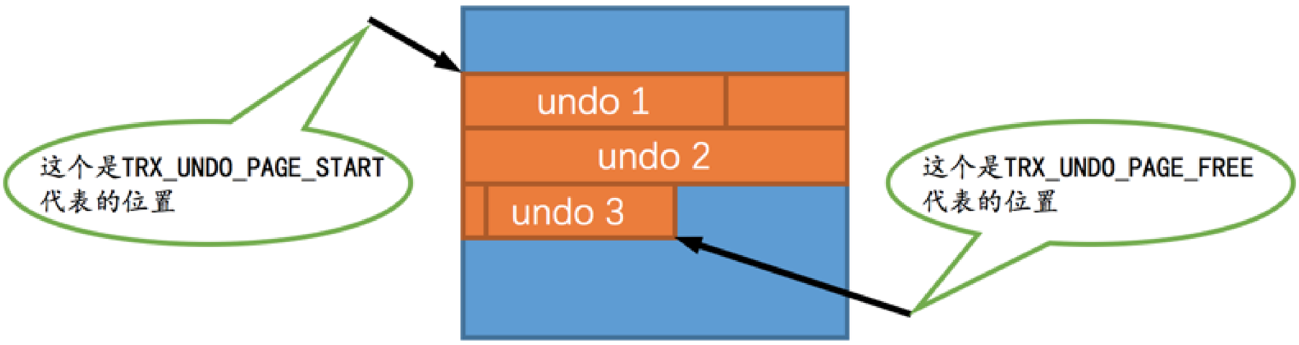
  - TRX_UNDO_INSERT （使用十进制 1 表示）：类型为 TRX_UNDO_INSERT_REC 的 undo日志 属于此大类，一般由 INSERT 语句产生，或者在 UPDATE 语句中有更新主键的情况也会产生此类型的 undo日志 。
  - TRX_UNDO_UPDATE （使用十进制 2 表示），除了类型为 TRX_UNDO_INSERT_REC 的 undo日志 ，其他类型的 undo日志 都属于这个大类，比如我们前边说的 TRX_UNDO_DEL_MARK_REC 、 TRX_UNDO_UPD_EXIST_REC 啥的，一般由 DELETE 、 UPDATE 语句产生的 undo日志 属于这个大类。

  这个 TRX_UNDO_PAGE_TYPE 属性可选的值就是上边的两个，用来标记本页面用于存储哪个大类的 undo日志 ，不同大类的 undo日志 不能混着存储，比如一个 Undo页面 的 TRX_UNDO_PAGE_TYPE 属性值为 TRX_UNDO_INSERT ，那么这个页面就只能存储类型为 TRX_UNDO_INSERT_REC 的 undo日志 ，其他类型的 undo日志 就不能放到这个页面中了。

  > 小贴士：
  > 之所以把undo日志分成两个大类，是因为类型为TRX_UNDO_INSERT_REC的undo日志在事务提交后可以直接删除掉，而其他类型的undo日志还需要为所谓的MVCC服务，不能直接删除掉，对它们的处理需要区别对待。当然，如果你看这段话迷迷糊糊的话，那就不需要再看一遍了，现在只需要知道undo日志分为2个大类就好了，更详细的东西我们后边会仔细唠叨的。

- TRX_UNDO_PAGE_START ：表示在当前页面中是从什么位置开始存储 undo日志 的，或者说表示第一条 undo日志 在本页面中的起始偏移量。
- TRX_UNDO_PAGE_FREE ：与上边的 TRX_UNDO_PAGE_START 对应，表示当前页面中存储的最后一条 undo 日志结束时的偏移量，或者说从这个位置开始，可以继续写入新的 undo日志 。

  假设现在向页面中写入了3条 undo日志 ，那么 TRX_UNDO_PAGE_START 和 TRX_UNDO_PAGE_FREE 的示意图就是这样：



当然，在最初一条 undo日志 也没写入的情况下， TRX_UNDO_PAGE_START 和 TRX_UNDO_PAGE_FREE 的值是相同的。

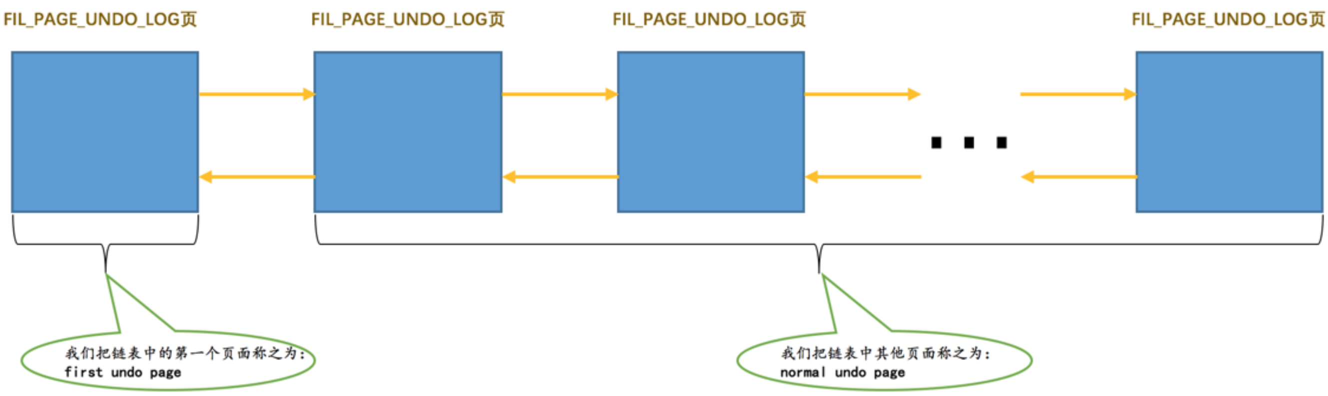
- TRX_UNDO_PAGE_NODE ：代表一个 List Node 结构（链表的普通节点，我们上边刚说的）。

  下边马上用到这个属性，稍安勿躁。

23.3 Undo页面链表

23.3.1 单个事务中的Undo页面链表

因为一个事务可能包含多个语句，而且一个语句可能对若干条记录进行改动，而对每条记录进行改动前，都需要记录1条或2条的 undo日志 ，所以在一个事务执行过程中可能产生很多 undo日志 ，这些日志可能一个页面放不下，需要放到多个页面中，这些页面就通过我们上边介绍的 TRX_UNDO_PAGE_NODE 属性连成了链表：

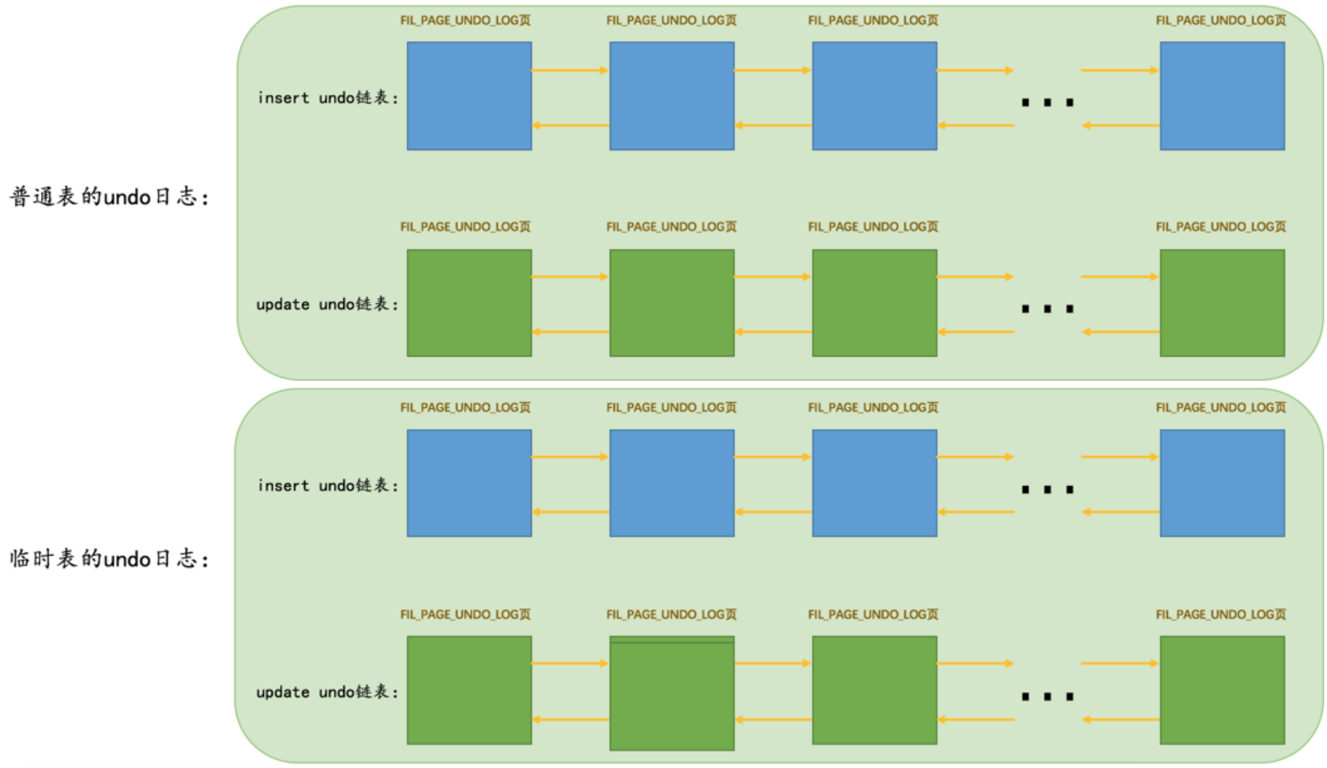


大家往上再瞅一瞅上边的图，我们特意把链表中的第一个 Undo页面 给标了出来，称它为 first undo page ，其余的 Undo页面 称之为 normal undo page ，这是因为在 first undo page 中除了记录 Undo Page Header 之外，还会记录其他的一些管理信息，这个我们稍后再说哈。

在一个事务执行过程中，可能混着执行 INSERT 、 DELETE 、 UPDATE 语句，也就意味着会产生不同类型的 undo日志 。但是我们前边又强调过，同一个 Undo页面 要么只存储 TRX_UNDO_INSERT 大类的 undo日志 ，要么只存储 TRX_UNDO_UPDATE 大类的 undo日志 ，反正不能混着存，所以在一个事务执行过程中就可能需要2个 Undo页面 的链表，一个称之为 insert undo链表 ，另一个称之为 update undo链表 ，画个示意图就是这样：



另外，设计 InnoDB 的大叔规定对普通表和临时表的记录改动时产生的 undo日志 要分别记录（我们稍后阐释为啥这么做），所以在一个事务中最多有4个以 Undo页面 为节点组成的链表：



当然，并不是在事务一开始就会为这个事务分配这4个链表，具体分配策略如下：

- 刚刚开启事务时，一个 Undo页面 链表也不分配。
- 当事务执行过程中向普通表中插入记录或者执行更新记录主键的操作之后，就会为其分配一个 普通表的insert undo链表 。
- 当事务执行过程中删除或者更新了普通表中的记录之后，就会为其分配一个 普通表的update undo链表 。
- 当事务执行过程中向临时表中插入记录或者执行更新记录主键的操作之后，就会为其分配一个 临时表的insert undo链表 。
- 当事务执行过程中删除或者更新了临时表中的记录之后，就会为其分配一个 临时表的update undo链表 。

总结一句就是：按需分配，啥时候需要啥时候再分配，不需要就不分配。

23.3.2 多个事务中的Undo页面链表

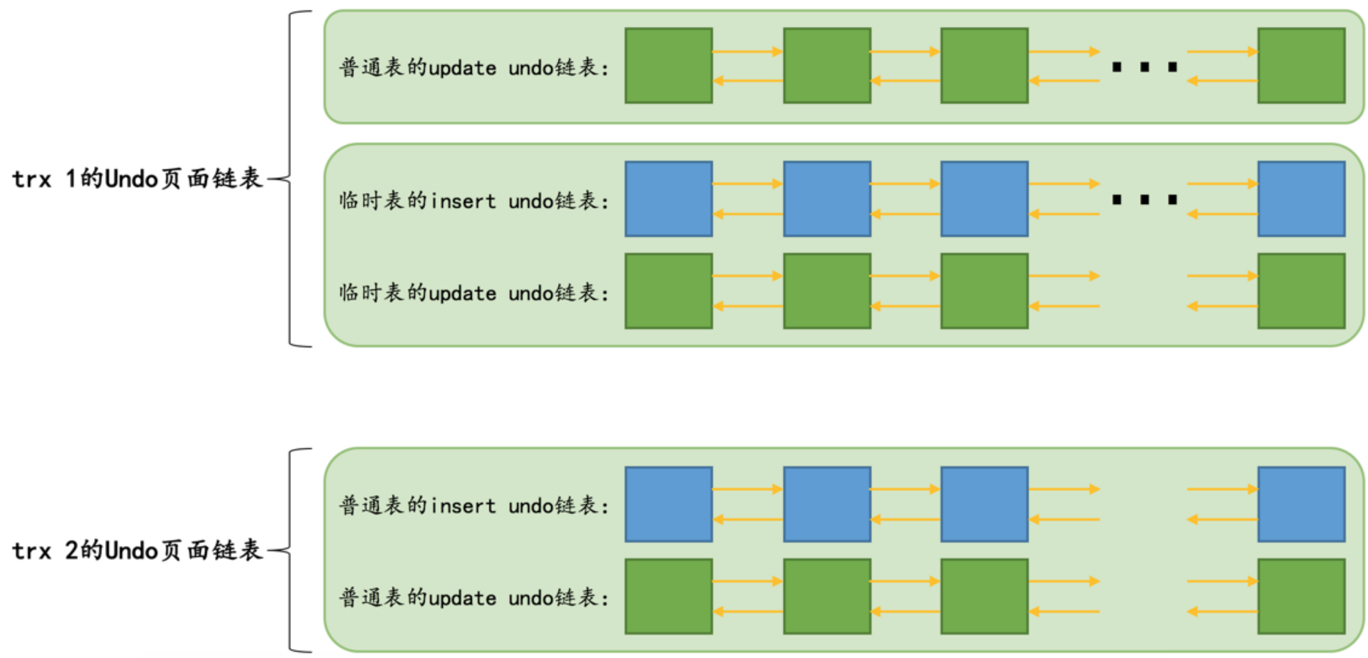
为了尽可能提高 undo日志 的写入效率，不同事务执行过程中产生的undo日志需要被写入到不同的Undo页面链表中。比方说现在有事务 id 分别为 1 、 2 的两个事务，我们分别称之为 trx 1 和 trx 2 ，假设在这两个事务执行过程中：

- trx 1 对普通表做了 DELETE 操作，对临时表做了 INSERT 和 UPDATE 操作。

  InnoDB 会为 trx 1 分配3个链表，分别是：
  - 针对普通表的 update undo链表
  - 针对临时表的 insert undo链表
  - 针对临时表的 update undo链表 。
- trx 2 对普通表做了 INSERT 、 UPDATE 和 DELETE 操作，没有对临时表做改动。

  InnoDB 会为 trx 2 分配2个链表，分别是：
  - 针对普通表的 insert undo链表
  - 针对普通表的 update undo链表 。

综上所述，在 trx 1 和 trx 2 执行过程中， InnoDB 共需为这两个事务分配5个 Undo页面 链表，画个图就是这样：

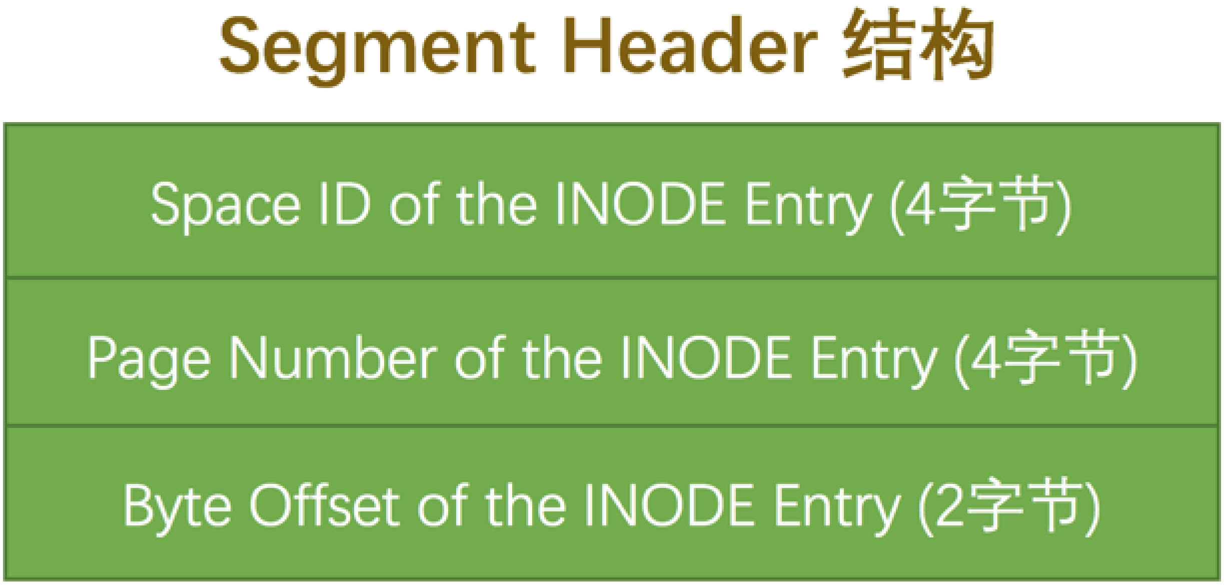


如果有更多的事务，那就意味着可能会产生更多的 Undo页面 链表。

23.4 undo日志具体写入过程

23.4.1 段（Segment）的概念

如果你有认真看过表空间那一章的话，对这个 段 的概念应该印象深刻，我们当时花了非常大的篇幅来唠叨这个概念。简单讲，这个 段 是一个逻辑上的概念，本质上是由若干个零散页面和若干个完整的区组成的。比如一个 B+ 树索引被划分成两个段，一个叶子节点段，一个非叶子节点段，这样叶子节点就可以被尽可能的存到一起，非叶子节点被尽可能的存到一起。每一个段对应一个 INODE Entry 结构，这个 INODE Entry 结构描述了这个段的各种信息，比如段的 ID ，段内的各种链表基节点，零散页面的页号有哪些等信息（具体该结构中每个属性的意思大家可以到表空间那一章里再次重温一下）。我们前边也说过，为了定位一个 INODE Entry ，设计 InnoDB 的大叔设计了一个 Segment Header 的结构：



整个 Segment Header 占用10个字节大小，各个属性的意思如下：

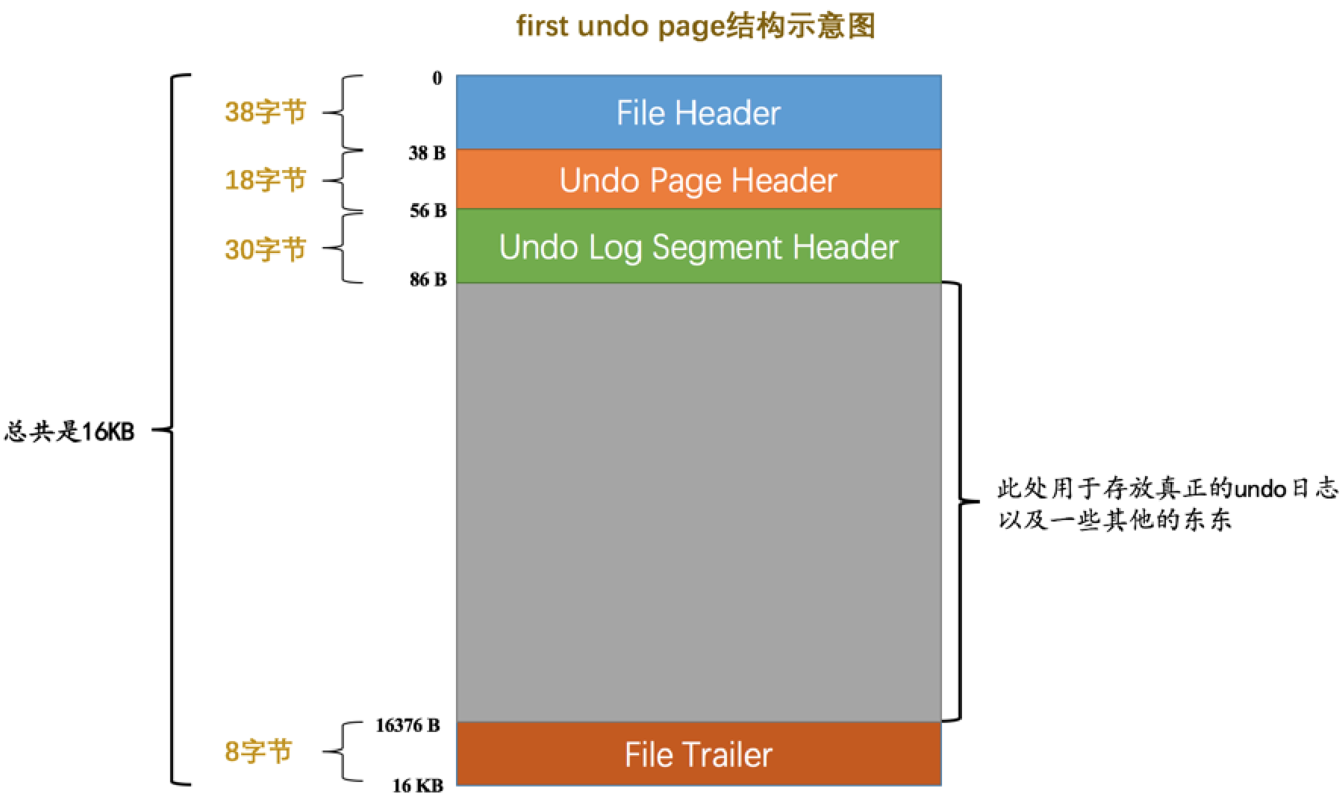
- Space ID of the INODE Entry ： INODE Entry 结构所在的表空间ID。
- Page Number of the INODE Entry ： INODE Entry 结构所在的页面页号。
- Byte Offset of the INODE Ent ： INODE Entry 结构在该页面中的偏移量

知道了表空间ID、页号、页内偏移量，不就可以唯一定位一个 INODE Entry 的地址了么~

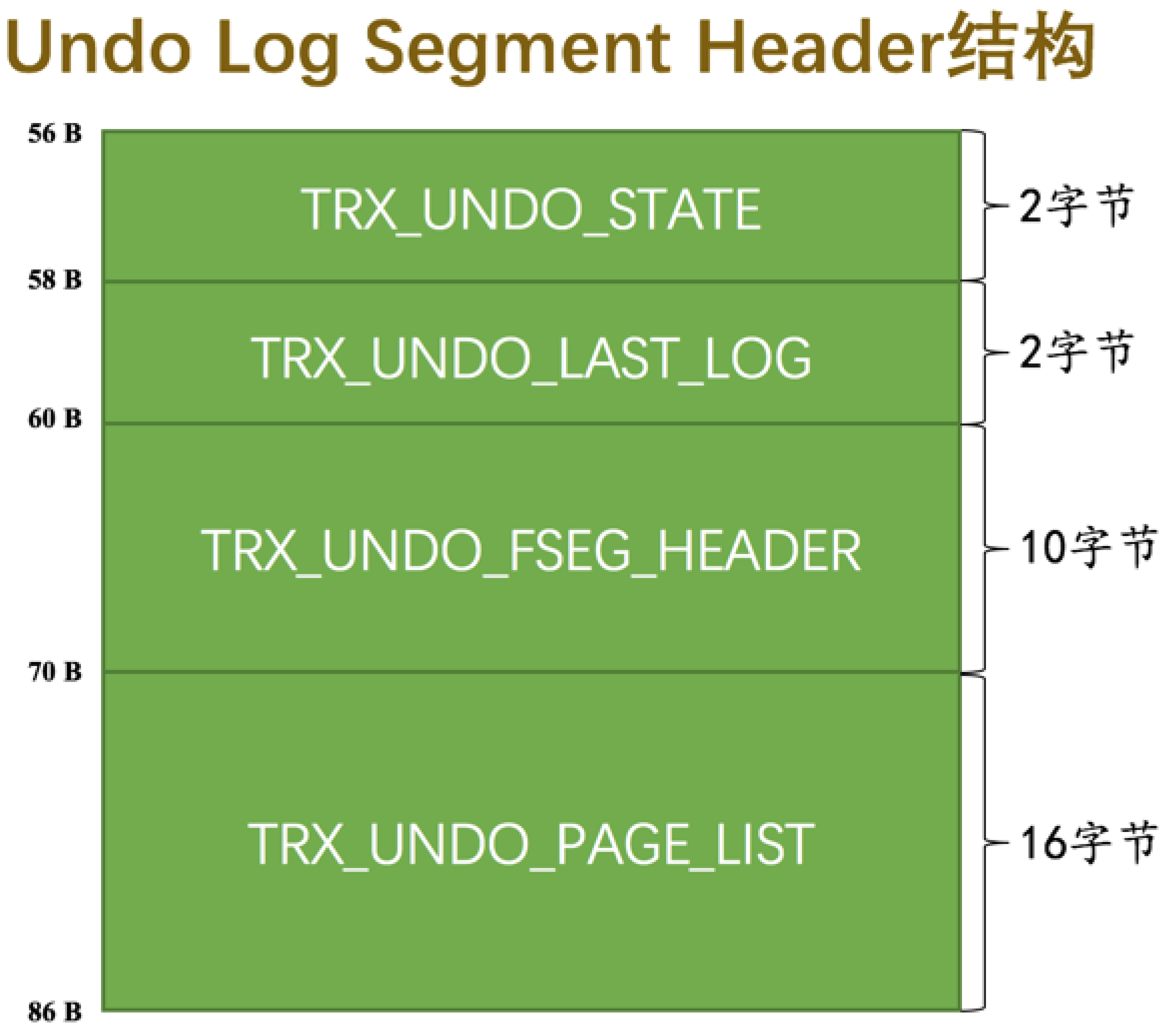
> 小贴士：
> 这部分关于段的各种概念我们在表空间那一章中都有详细解释，在这里重提一下只是为了唤醒大家沉睡的记忆，如果有任何不清楚的地方可以再次跳回表空间的那一章仔细读一下。

23.4.2 Undo Log Segment Header

设计 InnoDB 的大叔规定，每一个 Undo页面 链表都对应着一个 段 ，称之为 Undo Log Segment 。也就是说链表中的页面都是从这个段里边申请的，所以他们在 Undo页面 链表的第一个页面，也就是上边提到的 first undo page 中设计了一个称之为 Undo Log Segment Header 的部分，这个部分中包含了该链表对应的段的 segment header 信息以及其他的一些关于这个段的信息，所以 Undo 页面链表的第一个页面其实长这样：



可以看到这个 Undo 链表的第一个页面比普通页面多了个 Undo Log Segment Header ，我们来看一下它的结构：



其中各个属性的意思如下:

- TRX_UNDO_STATE : 本 Undo页面 链表处在什么状态。

  一个 Undo Log Segment 可能处在的状态包括:
  - TRX_UNDO_ACTIVE : 活跃状态，也就是一个活跃的事务正在往这个段里边写入 undo日志 。
  - TRX_UNDO_CACHED : 被缓存的状态。处在该状态的 Undo页面 链表等待着之后被其他事务重用。
  - TRX_UNDO_TO_FREE : 对于 insert undo 链表来说，如果在它对应的事务提交之后，该链表不能被重用，那么就会处于这种状态。
  - TRX_UNDO_TO_PURGE : 对于 update undo 链表来说，如果在它对应的事务提交之后，该链表不能被重用，那么就会处于这种状态。
  - TRX_UNDO_PREPARED : 包含处于 PREPARE 阶段的事务产生的 undo日志 。

  > 小贴士：
  > Undo页面链表什么时候会被重用，怎么重用我们之后会详细说的。事务的PREPARE阶段是在所谓的分布式事务中才出现的，本书中不会介绍更多关于分布式事务的事情，所以大家目前忽略这个状态就好了。

- TRX_UNDO_LAST_LOG : 本 Undo页面 链表中最后一个 Undo Log Header 的位置。

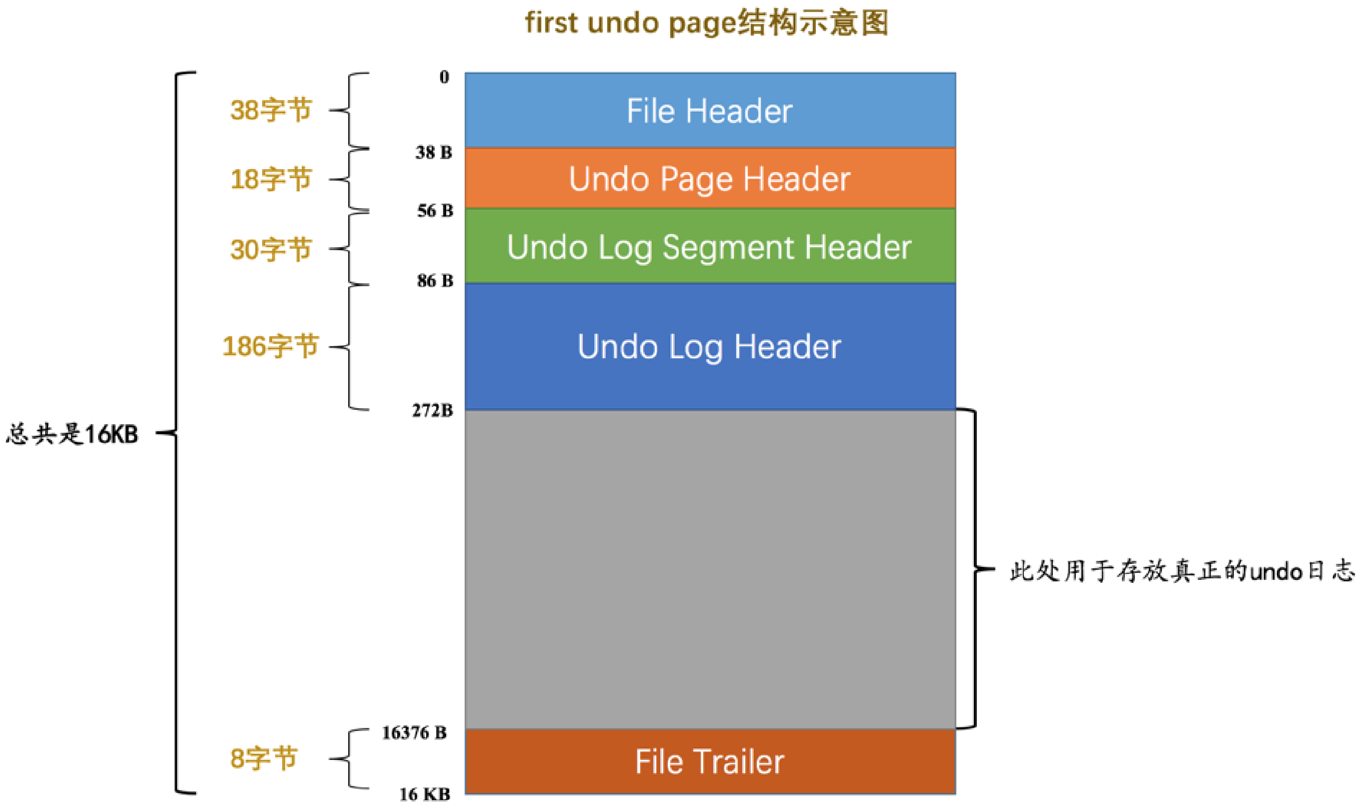
  > 小贴士：
  > 关于什么是Undo Log Header，我们稍后马上介绍哈。

- TRX_UNDO_FSEG_HEADER : 本 Undo页面 链表对应的段的 Segment Header 信息（就是我们上一节介绍的那个10字节结构，通过这个信息可以找到该段对应的 INODE Entry ）。
- TRX_UNDO_PAGE_LIST : Undo页面 链表的基节点。

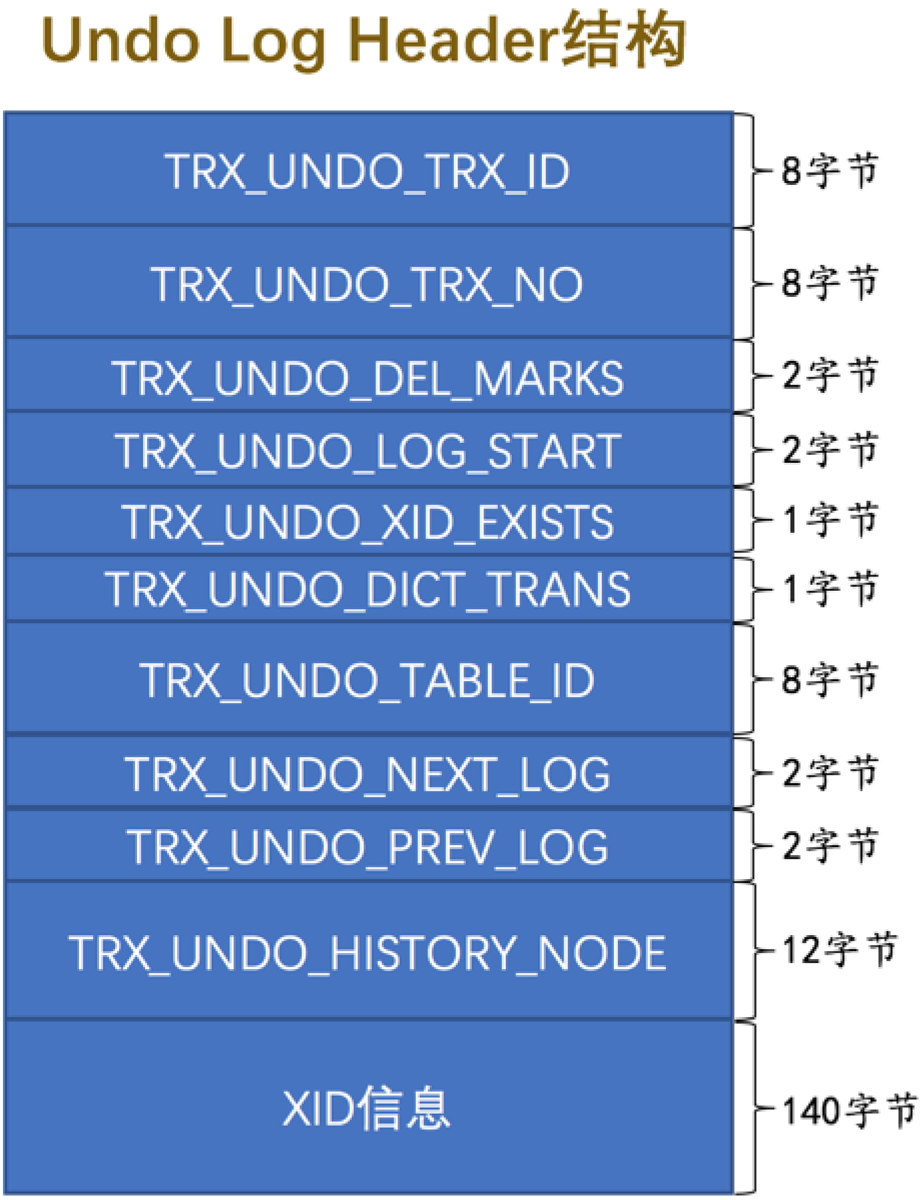
  我们上边说 Undo页面 的 Undo Page Header 部分有一个12字节大小的 TRX_UNDO_PAGE_NODE 属性，这个属性代表一个 List Node 结构。每一个 Undo页面 都包含 Undo Page Header 结构，这些页面就可以通过这个属性连成一个链表。这个 TRX_UNDO_PAGE_LIST 属性代表着这个链表的基节点，当然这个基节点只存在于 Undo页面 链表的第一个页面，也就是 first undo page 中。

23.4.3 Undo Log Header

一个事务在向 Undo页面 中写入 undo日志 时的方式是十分简单暴力的，就是直接往里怼，写完一条紧接着写另一条，各条 undo日志 之间是亲密无间的。写完一个 Undo页面 后，再从段里申请一个新页面，然后把这个页面插入到 Undo页面 链表中，继续往这个新申请的页面中写。设计 InnoDB 的大叔认为同一个事务向一个 Undo页面 链表中写入的 undo日志 算是一个组，比方说我们上边介绍的 trx 1 由于会分配3个 Undo页面 链表，也就会写入3个组的 undo日志 ； trx 2 由于会分配2个 Undo页面 链表，也就会写入2个组的 undo日志 。在每写入一组 undo日志 时，都会在这组 undo日志 前先记录一下关于这个组的一些属性，设计 InnoDB 的大叔把存储这些属性的地方称之为 Undo Log Header 。所以 Undo页面 链表的第一个页面在真正写入 undo日志 前，其实都会被填充 Undo Page Header 、 Undo Log Segment Header 、 Undo Log Header 这3个部分，如图所示:



这个 Undo Log Header 具体的结构如下：



哇唔，映入眼帘的又是一大坨属性，我们先大致看一下它们都是啥意思：

- `TRX_UNDO_TRX_ID` ：生成本组 `undo日志` 的事务 `id` 。
- `TRX_UNDO_TRX_NO` ：事务提交后生成的一个需要序号，使用此序号来标记事务的提交顺序（先提交的此序号小，后提交的此序号大）。
- `TRX_UNDO_DEL_MARKS` ：标记本组 `undo` 日志中是否包含由于 `Delete mark` 操作产生的 `undo日志` 。
- `TRX_UNDO_LOG_START` ：表示本组 `undo` 日志中第一条 `undo日志` 的在页面中的偏移量。
- `TRX_UNDO_XID_EXISTS` ：本组 `undo日志` 是否包含XID信息。

> 小贴士：
> 本书不会讲述更多关于XID是个什么东东，有兴趣的同学可以到搜索引擎或者文档中搜一搜哈。

- `TRX_UNDO_DICT_TRANS` ：标记本组 `undo日志` 是不是由DDL语句产生的。
- `TRX_UNDO_TABLE_ID` ：如果 `TRX_UNDO_DICT_TRANS` 为真，那么本属性表示DDL语句操作的表的 `table id` 。
- `TRX_UNDO_NEXT_LOG` ：下一组的 `undo日志` 在页面中开始的偏移量。
- `TRX_UNDO_PREV_LOG` ：上一组的 `undo日志` 在页面中开始的偏移量。

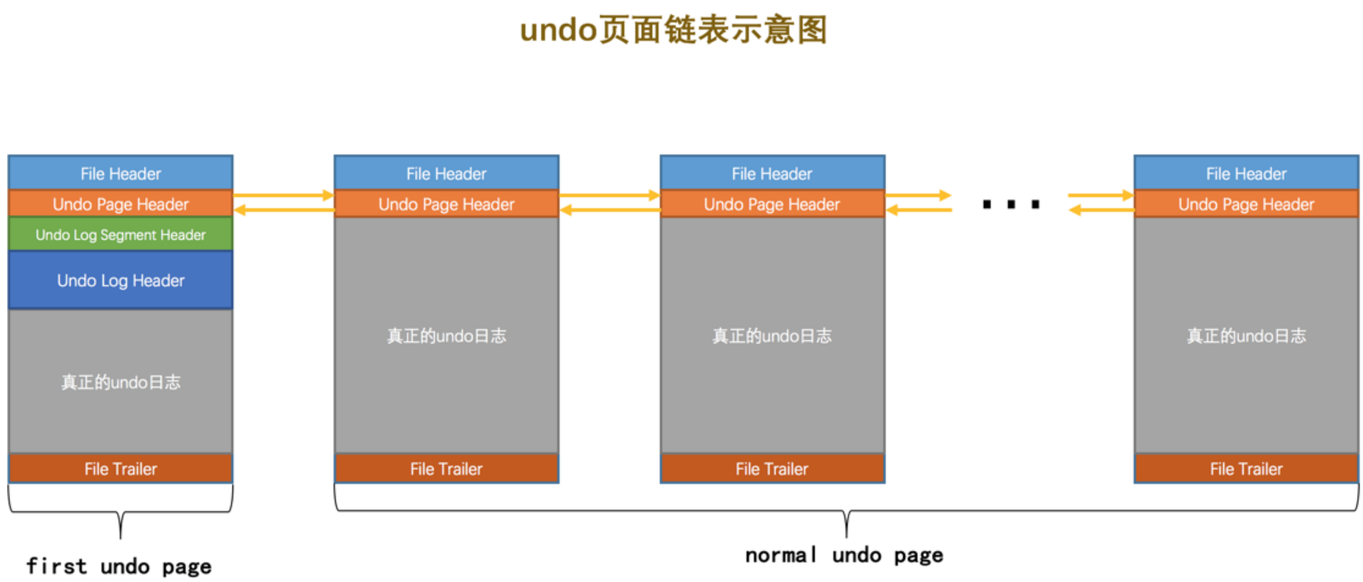
> 小贴士：
>
> 一般来说一个Undo页面链表只存储一个事务执行过程中产生的一组undo日志，但是在某些情况下，可能会在一个事务提交之后，之后开启的事务重复利用这个Undo页面链表，这样就会导致一个Undo页面中可能存放多组Undo日志，TRX_UNDO_NEXT_LOG和TRX_UNDO_PREV_LOG就是用来标记下一组和上一组undo日志在页面中的偏移量的。关于什么时候重用Undo页面链表，怎么重用这个链表我们稍后会详细说明的，现在先理解TRX_UNDO_NEXT_LOG和TRX_UNDO_PREV_LOG这两个属性的意思就好了。

- TRX_UNDO_HISTORY_NODE ：一个12字节的 List Node 结构，代表一个称之为 History 链表的节点。

> 小贴士：
>
> 关于History链表我们后边会格外详细的唠叨，现在先不用管哈。

23.4.4 小结

对于没有被重用的 Undo页面 链表来说，链表的第一个页面，也就是 first undo page 在真正写入 undo日志 前，会填充 Undo Page Header 、 Undo Log Segment Header 、 Undo Log Header 这3个部分，之后才开始正式写入 undo日志 。对于其他的页面来说，也就是 normal undo page 在真正写入 undo日志 前，只会填充 Undo Page Header 。链表的 List Base Node 存放到 first undo page 的 Undo Log Segment Header 部分， List Node 信息存放到每一个 Undo页面 的 undo Page Header 部分，所以画一个 Undo页面 链表的示意图就是这样：



23.5 重用Undo页面

我们前边说为了能提高并发执行的多个事务写入 undo日志 的性能，设计 InnoDB 的大叔决定为每个事务单独分配相应的 Undo页面 链表（最多可能单独分配4个链表）。但是这样也造成了一些问题，比如其实大部分事务执行过程中可能只修改了一条或几条记录，针对某个 Undo页面 链表只产生了非常少的 undo日志 ，这些 undo日志 可能只占用一丢丢存储空间，每开启一个事务就新创建一个 Undo页面 链表（虽然这个链表中只有一个页面）来存储这么一丢丢 undo日志 岂不是太浪费了么？的确是挺浪费，于是设计 InnoDB 的大叔本着勤俭节约的优良传统，决定在事务提交后在某些情况下重用该事务的 Undo页面 链表。一个 Undo页面 链表是否可以被重用的条件很简单：

- 该链表中只包含一个 Undo页面 。

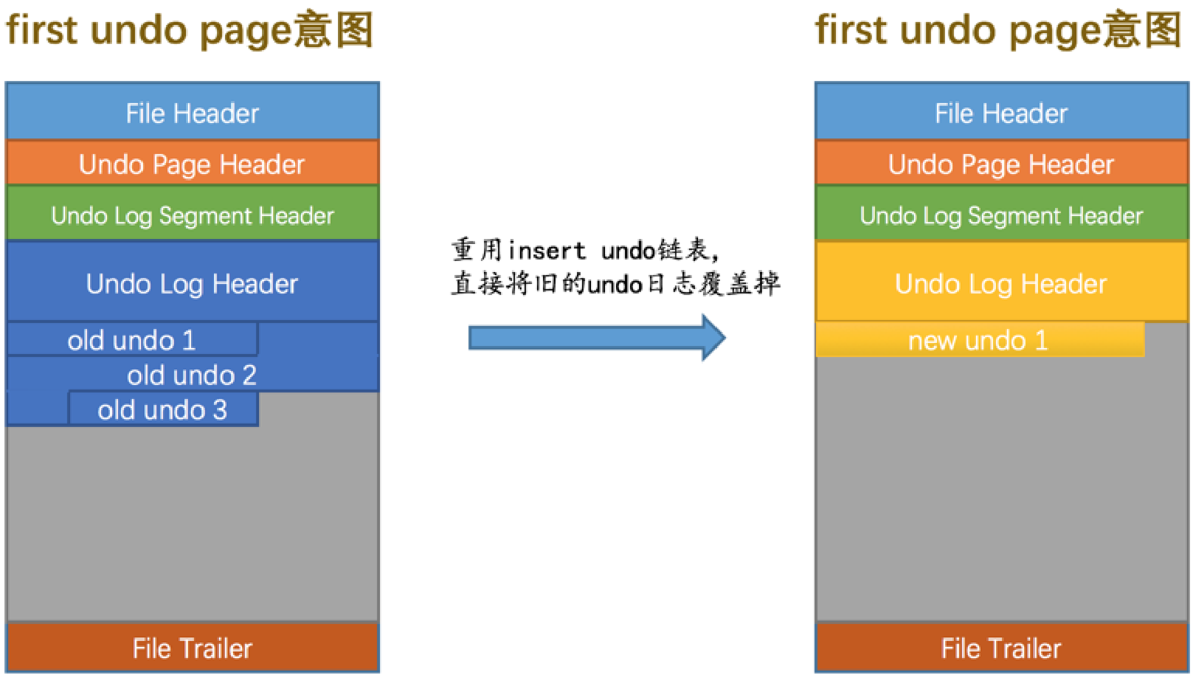
  如果一个事务执行过程中产生了非常多的 undo日志 ，那么它可能申请非常多的页面加入到 Undo页面 链表中。在该事物提交后，如果将整个链表中的页面都重用，那就意味着即使新的事务并没有向该 Undo页面 链表中写入很多 undo日志 ，那该链表中也得维护非常多的页面，那些用不到的页面也不能被别的事务所使用，这样就造成了另一种浪费。所以设计 InnoDB 的大叔们规定，只有在 Undo页面 链表中只包含一个 Undo页面 时，该链表才可以被下一个事务所重用。

- 该 Undo页面 已经使用的空间小于整个页面空间的3/4。

我们前边说过， Undo页面 链表按照存储的 undo日志 所属的大类可以被分为 insert undo链表 和 update undo 链表 两种，这两种链表在被重用时的策略也是不同的，我们分别看一下：

- insert undo链表

 insert undo链表 中只存储类型为 TRX_UNDO_INSERT_REC 的 undo日志 ，这种类型的 undo日志 在事务提交之后就没用了，就可以被清除掉。所以在某个事务提交后，重用这个事务的 insert undo链表 （这个链表中只有一个页面）时，可以直接把之前事务写入的一组 undo日志 覆盖掉，从头开始写入新事务的一组 undo日志 ，如下图所示：

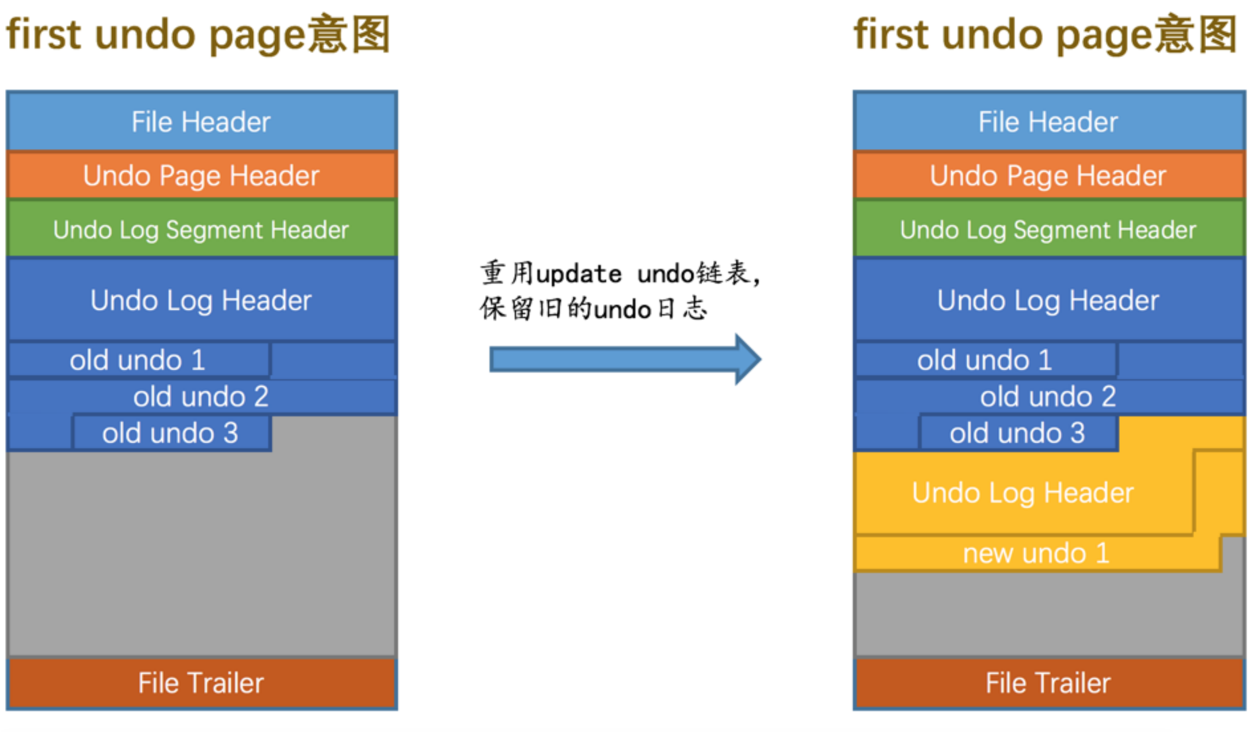


如图所示，假设有一个事务使用的 insert undo链表 ，到事务提交时，只向 insert undo链表 中插入了3条 undo日志 ，这个 insert undo链表 只申请了一个 Undo页面 。假设此刻该页面已使用的空间小于整个页面大小的3/4，那么下一个事务就可以重用这个 insert undo链表 （链表中只有一个页面）。假设此时有一个新事务重用了该 insert undo链表 ，那么可以直接把旧的一组 undo日志 覆盖掉，写入一组新的 undo日志 。

> 小贴士：
> 当然，在重用Undo页面链表写入新的一组undo日志时，不仅会写入新的Undo Log Header，还会适当调整Undo Page Header、Undo Log Segment Header、Undo Log Header中的一些属性，比如TRX_UNDO_PAGE_START、TRX_UNDO_PAGE_FREE等等等等，这些我们就不具体唠叨了。

- update undo链表

在一个事务提交后，它的 update undo链表 中的 undo日志 也不能立即删除掉（这些日志用于MVCC，我们后边会说的）。所以如果之后的事务想重用 update undo链表 时，就不能覆盖之前事务写入的 undo日志 。这样就相当于在同一个 Undo页面 中写入了多组的 undo日志 ，效果看起来就是这样：

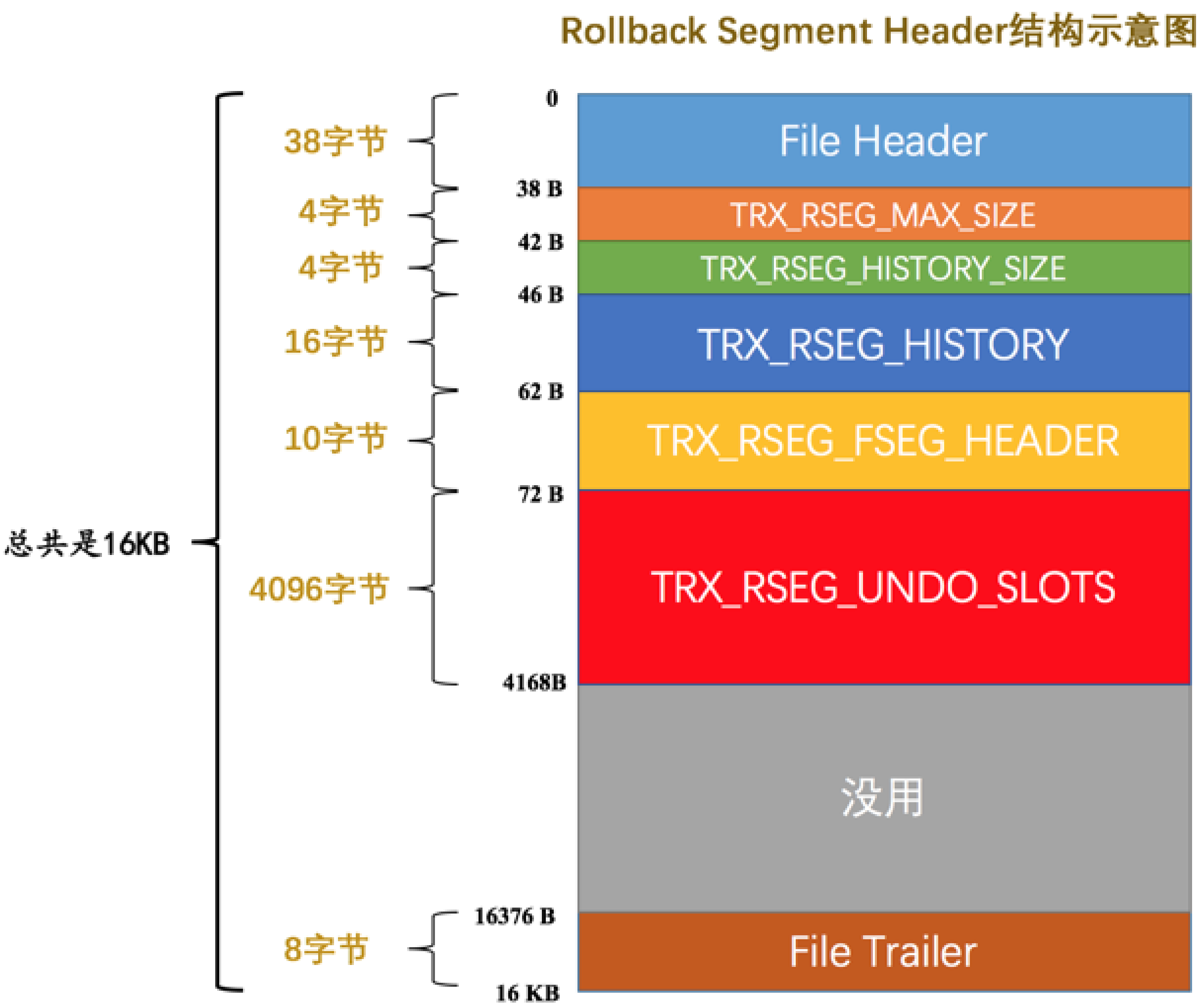


23.6 回滚段

23.6.1 回滚段的概念

我们现在知道一个事务在执行过程中最多可以分配4个 Undo页面 链表，在同一时刻不同事务拥有的 Undo页面 链表是不一样的，所以在同一时刻系统里其实可以有许许多多个 Undo页面 链表存在。为了更好的管理这些链表，设计 InnoDB 的大叔又设计了一个称之为 Rollback Segment Header 的页面，在这个页面中存放了各个 Undo页面 链表的 frist undo page 的 页号 ，他们把这些 页号 称之为 undo slot 。我们可以这样理解，每个 Undo页面 链表都相当于是一个班，这个链表的 first undo page 就相当于这个班的班长，找到了这个班的班长，就可以找到班里的其他同学（其他同学相当于 normal undo page ）。有时候学校需要向这些班级传达一下精神，就需要把班长都召集在会议室，这个 Rollback Segment Header 就相当于是一个会议室。

我们看一下这个称之为 Rollback Segment Header 的页面长啥样（以默认的16KB为例）：



设计 InnoDB 的大叔规定，每一个 Rollback Segment Header 页面都对应着一个段，这个段就称为 Rollback Segment ，翻译过来就是 回滚段 。与我们之前介绍的各种段不同的是，这个 Rollback Segment 里其实只有一个页面（这可能是设计 InnoDB 的大叔们的一种洁癖，他们可能觉得为了某个目的去分配页面的话都得先申请一个段，或者他们觉得虽然目前版本的 MySQL 里 Rollback Segment 里其实只有一个页面，但可能之后的版本里会增加页面也说不定）。

了解了 Rollback Segment 的含义之后，我们再来看看这个称之为 Rollback Segment Header 的页面的各个部分的含义都是啥意思：

- TRX_RSEG_MAX_SIZE ：本 Rollback Segment 中管理的所有 Undo页面 链表中的 Undo页面 数量之和的最大值。换句话说，本 Rollback Segment 中所有 Undo页面 链表中的 Undo页面 数量之和不能超过 TRX_RSEG_MAX_SIZE 代表的值。

  该属性的值默认为无限大，也就是我们想写多少 Undo页面 都可以。

  > 小贴士：
  > 无限大其实也只是个夸张的说法，4个字节能表示最大的数也就是0xFFFFFFFF，但是我们之后会看到，0xFFFFFFFF这个数有特殊用途，所以实际上TRX_RSEG_MAX_SIZE的值为0xFFFFFFFE。

- TRX_RSEG_HISTORY_SIZE ： History 链表占用的页面数量。
- TRX_RSEG_HISTORY ： History 链表的基节点。

  > 小贴士：
  > History链表后边讲，稍安勿躁。

- TRX_RSEG_FSEG_HEADER ：本 Rollback Segment 对应的10字节大小的 Segment Header 结构，通过它可以找到本段对应的 INODE Entry 。
- TRX_RSEG_UNDO_SLOTS ：各个 Undo页面 链表的 first undo page 的 页号 集合，也就是 undo slot 集合。

  一个页号占用 4 个字节，对于 16KB 大小的页面来说，这个 TRX_RSEG_UNDO_SLOTS 部分共存储了 1024 个 undo slot ，所以共需 1024 × 4 = 4096 个字节。

23.6.2 从回滚段中申请Undo页面链表

初始情况下，由于未向任何事务分配任何 Undo页面 链表，所以对于一个 Rollback Segment Header 页面来说，它的各个 undo slot 都被设置成了一个特殊的值： FIL_NULL （对应的十六进制就是 0xFFFFFFFF ），表示该 undo slot 不指向任何页面。

随着时间的流逝，开始有事务需要分配 Undo页面 链表了，就从回滚段的第一个 undo slot 开始，看看该 undo slot 的值是不是 FIL_NULL ：

- 如果是 FIL_NULL ，那么在表空间中新创建一个段（也就是 Undo Log Segment ），然后从段里申请一个页面作为 Undo页面 链表的 first undo page ，然后把该 undo slot 的值设置为刚刚申请的这个页面的地址，这样也就意味着这个 undo slot 被分配给了这个事务。
- 如果不是 FIL_NULL ，说明该 undo slot 已经指向了一个 undo链表 ，也就是说这个 undo slot 已经被别的事务占用了，那就跳到下一个 undo slot ，判断该 undo slot 的值是不是 FIL_NULL ，重复上边的步骤。

一个 Rollback Segment Header 页面中包含 1024 个 undo slot ，如果这 1024 个 undo slot 的值都不为 FIL_NULL ，这就意味着这 1024 个 undo slot 都已经名花有主（被分配给了某个事务），此时由于新事务无法再获得新的 Undo页面 链表，就会回滚这个事务并且给用户报错：

```
Too many active concurrent transactions
```

用户看到这个错误，可以选择重新执行这个事务（可能重新执行时有别的事务提交了，该事务就可以被分配 Undo页面 链表了）。

当一个事务提交时，它所占用的 undo slot 有两种命运：

- 如果该 undo slot 指向的 Undo页面 链表符合被重用的条件（就是我们上边说的 Undo页面 链表只占用一个页面并且已使用空间小于整个页面的3/4）。

  该 undo slot 就处于被缓存的状态，设计 InnoDB 的大叔规定这时该 Undo页面 链表的 TRX_UNDO_STATE 属性（该属性在 first undo page 的 Undo Log Segment Header 部分）会被设置为 TRX_UNDO_CACHED 。

  被缓存的 undo slot 都会被加入到一个链表，根据对应的 Undo页面 链表的类型不同，也会被加入到不同的链表：
  - 如果对应的 Undo页面 链表是 insert undo链表 ，则该 undo slot 会被加入 insert undo cached链表 。
  - 如果对应的 Undo页面 链表是 update undo链表 ，则该 undo slot 会被加入 update undo cached链表 。

    一个回滚段就对应着上述两个 cached链表 ，如果有新事务要分配 undo slot 时，先从对应的 cached链表 中找。如果没有被缓存的 undo slot ，才会到回滚段的 Rollback Segment Header 页面中再去找。
- 如果该 undo slot 指向的 Undo页面 链表不符合被重用的条件，那么针对该 undo slot 对应的 Undo页面 链表类型不同，也会有不同的处理：
  - 如果对应的 Undo页面 链表是 insert undo链表 ，则该 Undo页面 链表的 TRX_UNDO_STATE 属性会被设置为 TRX_UNDO_TO_FREE ，之后该 Undo页面 链表对应的段会被释放掉（也就意味着段中的页面可以被挪作他用），然后把该 undo slot 的值设置为 FIL_NULL 。

- 如果对应的 Undo页面 链表是 update undo链表 ，则该 Undo页面 链表的 TRX_UNDO_STATE 属性会被设置为 TRX_UNDO_TO_PRUGE ，则会将该 undo slot 的值设置为 FIL_NULL ，然后将本次事务写入的一组 undo 日志放到所谓的 History链表 中（需要注意的是，这里并不会将 Undo页面 链表对应的段给释放掉，因为这些 undo 日志还有用呢～）。

> 小贴士：
> 更多关于History链表的事我们稍后再说，稍安勿躁哈。

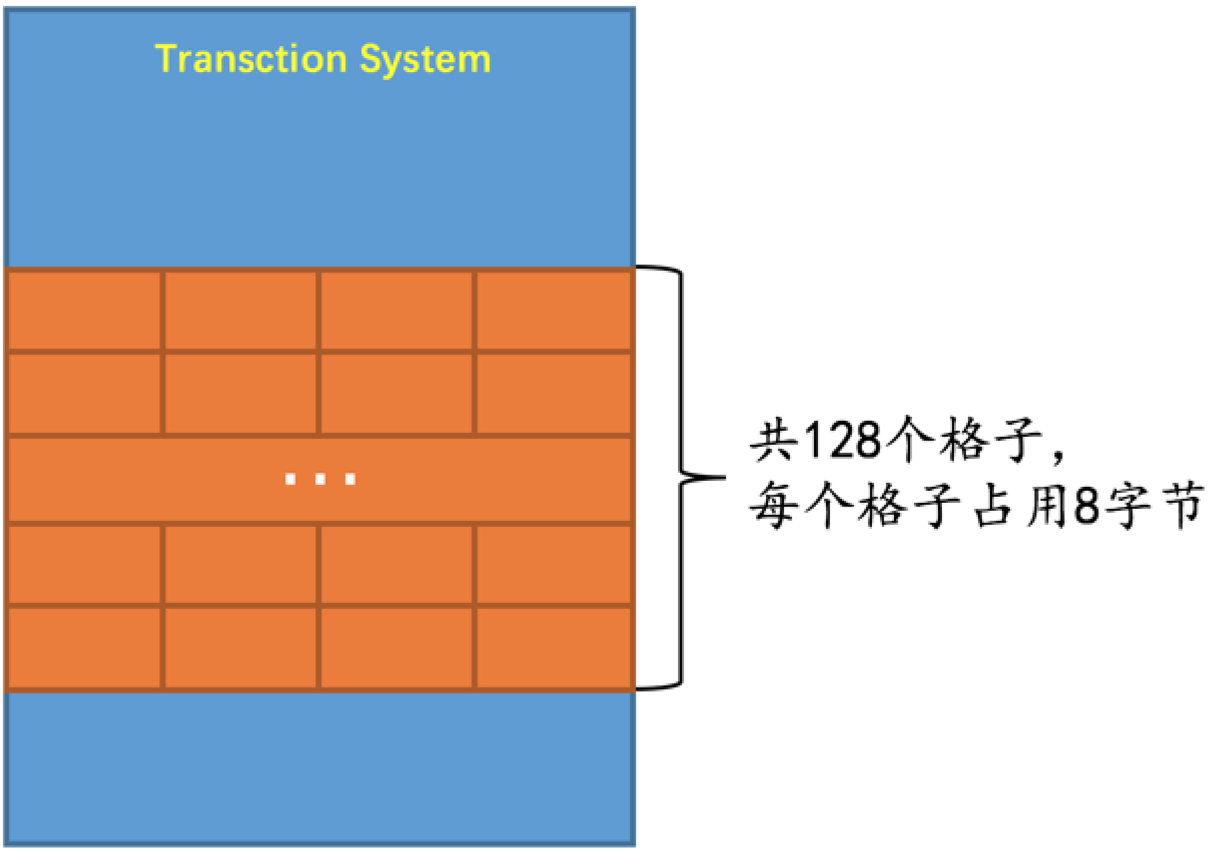
23.6.3 多个回滚段

我们说一个事务执行过程中最多分配 4 个 Undo页面 链表，而一个回滚段里只有 1024 个 undo slot ，很显然 undo slot 的数量有点少啊。我们即使假设一个读写事务执行过程中只分配 1 个 Undo页面 链表，那 1024 个 undo slot 也只能支持 1024 个读写事务同时执行，再多了就崩溃了。这就相当于会议室只能容下1024个班长同时开会，如果有几千人同时到会议室开会的话，那后来的那些班长就没地方坐了，只能等待前边的人开完会自己再进去开。

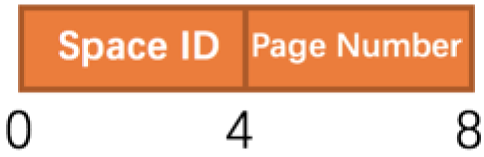
话说在 InnoDB 的早期发展阶段的确只有一个回滚段，但是设计 InnoDB 的大叔后来意识到了这个问题，咋解决这问题呢？会议室不够，多盖几个会议室不就得了。所以设计 InnoDB 的大叔一口气定义了 128 个回滚段，也就相当于有了 128 × 1024 = 131072 个 undo slot 。假设一个读写事务执行过程中只分配 1 个 Undo页面 链表，那么就可以同时支持 131072 个读写事务并发执行（这么多事务在一台机器上并发执行，还真没见过呢～）。

> 小贴士：
> 只读事务并不需要分配Undo页面链表，MySQL 5.7中所有刚开启的事务默认都是只读事务，只有在事务执行过程中对记录做了某些改动时才会被升级为读写事务。

每个回滚段都对应着一个 Rollback Segment Header 页面，有128个回滚段，自然就要有128个 Rollback Segment Header 页面，这些页面的地址总得找个地方存一下吧！于是设计 InnoDB 的大叔在系统表空间的第 5 号页面的某个区域包含了128个8字节大小的格子：



每个8字节的格子的构造就像这样：

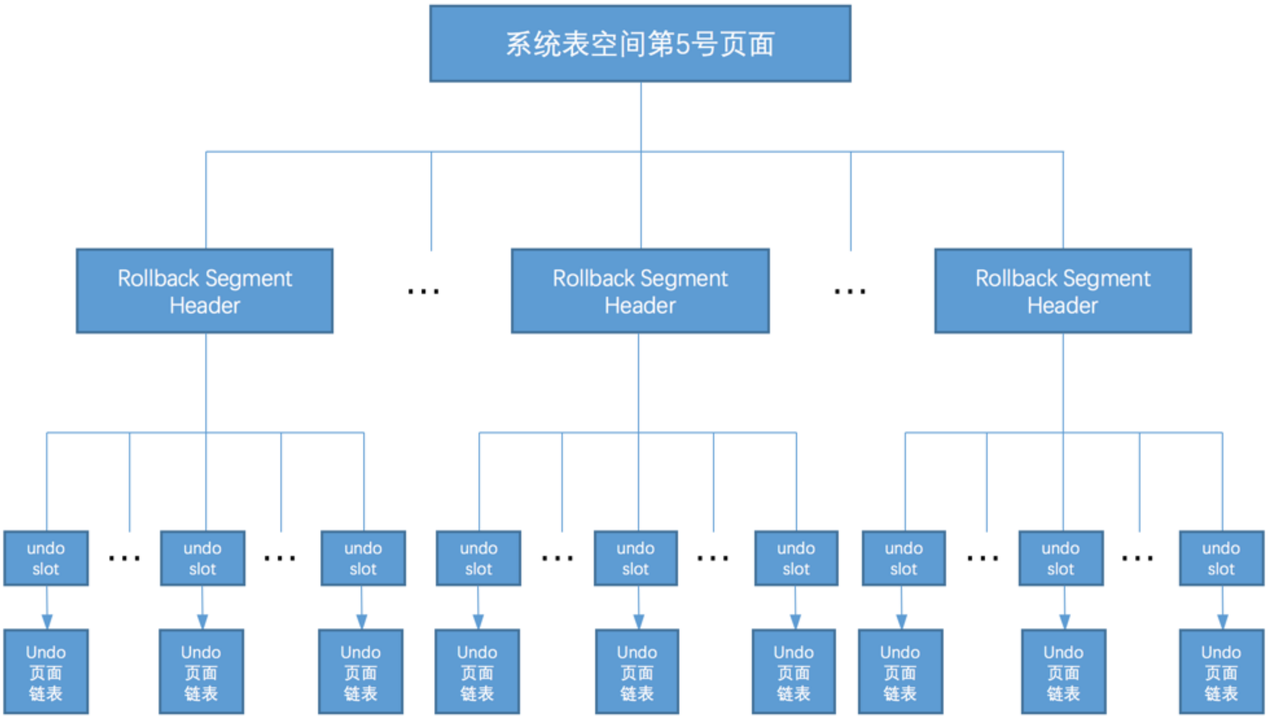


如果所示，每个8字节的格子其实由两部分组成：

- 4字节大小的 Space ID ，代表一个表空间的ID。
- 4字节大小的 Page number ，代表一个页号。

也就是说每个8字节大小的 格子 相当于一个指针，指向某个表空间中的某个页面，这些页面就是 Rollback Segment Header 。这里需要注意的一点事，要定位一个 Rollback Segment Header 还需要知道对应的表空间ID，这也就意味着不同的回滚段可能分布在不同的表空间中。

所以通过上边的叙述我们可以大致清楚，在系统表空间的第 5 号页面中存储了128个 Rollback Segment Header 页面地址，每个 Rollback Segment Header 就相当于一个回滚段。在 Rollback Segment Header 页面中，又包含 1024 个 undo slot ，每个 undo slot 都对应一个 Undo页面 链表。我们画个示意图：



把图一画出来就清爽多了。

23.6.4 回滚段的分类

我们把这128个回滚段给编一下号，最开始的回滚段称之为 第0号回滚段 ，之后依次递增，最后一个回滚段就称之为 第127号回滚段 。这128个回滚段可以被分成两大类：

- 第 0 号、第 33～127 号回滚段属于一类。其中第 0 号回滚段必须在系统表空间中（就是说第 0 号回滚段对应的 Rollback Segment Header 页面必须在系统表空间中），第 33～127 号回滚段既可以在系统表空间中，也可以在自己配置的 undo 表空间中，关于怎么配置我们稍后再说。

  如果一个事务在执行过程中由于对普通表的记录做了改动需要分配 Undo页面 链表时，必须从这一类的段中分配相应的 undo slot 。
- 第 1～32 号回滚段属于一类。这些回滚段必须在临时表空间（对应着数据目录中的 ibtmp1 文件）中。

  如果一个事务在执行过程中由于对临时表的记录做了改动需要分配 Undo页面 链表时，必须从这一类的段中分配相应的 undo slot 。

也就是说如果一个事务在执行过程中既对普通表的记录做了改动，又对临时表的记录做了改动，那么需要为这个记录分配2个回滚段，再分别到这两个回滚段中分配对应的 undo slot 。

不知道大家有没有疑惑，为啥要把针对普通表和临时表来划分不同种类的 回滚段 呢？这个还得从 Undo页面 本身说起，我们说 Undo页面 其实是类型为 FIL_PAGE_UNDO_LOG 的页面的简称，说到底它也是一个普通的页面。我们前边说过，在修改页面之前一定要先把对应的 redo日志 写上，这样在系统奔溃重启时才能恢复到奔溃前的状态。我们向 Undo页面 写入 undo日志 本身也是一个写页面的过程，设计 InnoDB 的大叔为此还设计了许多种 redo日志 的类型，比方说 MLOG_UNDO_HDR_CREATE 、 MLOG_UNDO_INSERT 、 MLOG_UNDO_INIT 等等等等，也就是说我们对 Undo页面 做的任何改动都会记录相应类型的 redo日志 。但是对于临时表来说，因为修改临时表而产生的 undo日志 只需要在系统运行过程中有效，如果系统奔溃了，那么在重启时也不需要恢复这些 undo 日志所在的页面，所以在写针对临时表的 Undo页面 时，并不需要记录相应的 redo日志 。总结一下针对普通表和临时表划分不同种类的 回滚段 的原因：在修改针对普通表的回滚段中的Undo页面时，需要记录对应的redo日志，而修改针对临时表的回滚段中的Undo页面时，不需要记录对应的redo日志。

> 小贴士：
> 实际上在MySQL 5.7.21这个版本中，如果我们仅仅对普通表的记录做了改动，那么只会为该事务分配针对普通表的回滚段，不分配针对临时表的回滚段。但是如果我们仅仅对临时表的记录做了改动，那么既会为该事务分配针对普通表的回滚段，又会为其分配针对临时表的回滚段（不过分配了回滚段并不会立即分配undo slot，只有在真正需要Undo页面链表时才会去分配回滚段中的undo slot）。

23.6.5 为事务分配Undo页面链表详细过程

上边说了一大堆的概念，大家应该有一点点的小晕，接下来我们以事务对普通表的记录做改动为例，给大家梳理一下事务执行过程中分配 Undo页面 链表时的完整过程，

- 事务在执行过程中对普通表的记录首次做改动之前，首先会到系统表空间的第 5 号页面中分配一个回滚段（其实就是获取一个 Rollback Segment Header 页面的地址）。一旦某个回滚段被分配给了这个事务，那么之后该事务中再对普通表的记录做改动时，就不会重复分配了。

  使用传说中的 round-robin （循环使用）方式来分配回滚段。比如当前事务分配了第 0 号回滚段，那么下一个事务就要分配第 33 号回滚段，下下个事务就要分配第 34 号回滚段，简单一点的说就是这些回滚段被轮着分配给不同的事务（就是这么简单粗暴，没啥好说的）。
- 在分配到回滚段后，首先看一下这个回滚段的两个 cached链表 有没有已经缓存了的 undo slot ，比如如果事务做的是 INSERT 操作，就去回滚段对应的 insert undo cached链表 中看看有没有缓存的 undo slot ；如果事务做的是 DELETE 操作，就去回滚段对应的 update undo cached链表 中看看有没有缓存的 undo slot 。如果有缓存的 undo slot ，那么就把这个缓存的 undo slot 分配给该事务。
- 如果没有缓存的 undo slot 可供分配，那么就要到 Rollback Segment Header 页面中找一个可用的 undo slot 分配给当前事务。

  从 Rollback Segment Header 页面中分配可用的 undo slot 的方式我们上边也说过了，就是从第 0 个 undo slot 开始，如果该 undo slot 的值为 FIL_NULL ，意味着这个 undo slot 是空闲的，就把这个 undo slot 分配给当前事务，否则查看第 1 个 undo slot 是否满足条件，依次类推，直到最后一个 undo slot 。如果这 1024 个 undo slot 都没有值为 FIL_NULL 的情况，就直接报错喽（一般不会出现这种情况）~

- 找到可用的 undo slot 后，如果该 undo slot 是从 cached链表 中获取的，那么它对应的 Undo Log Segment 已经分配了，否则的话需要重新分配一个 Undo Log Segment ，然后从该 Undo Log Segment 中申请一个页面作为 Undo页面 链表的 first undo page 。
- 然后事务就可以把 undo日志 写入到上边申请的 Undo页面 链表了！

对临时表的记录做改动的步骤和上述的一样，就不赘述了。不错需要再次强调一次，如果一个事务在执行过程中既对普通表的记录做了改动，又对临时表的记录做了改动，那么需要为这个记录分配2个回滚段。并发执行的不同事务其实也可以被分配相同的回滚段，只要分配不同的undo slot就可以了。

23.7 回滚段相关配置

23.7.1 配置回滚段数量

我们前边说系统中一共有 128 个回滚段，其实这只是默认值，我们可以通过启动参数 innodb_rollback_segments 来配置回滚段的数量，可配置的范围是 1~128 。但是这个参数并不会影响针对临时表的回滚段数量，针对临时表的回滚段数量一直是 32 ，也就是说：

- 如果我们把 innodb_rollback_segments 的值设置为 1 ，那么只会有1个针对普通表的可用回滚段，但是仍然有32个针对临时表的可用回滚段。
-
    - 如果我们把 innodb_rollback_segments 的值设置为 2～33 之间的数，效果和将其设置为 1 是一样的。
- 如果我们把 innodb_rollback_segments 设置为大于 33 的数，那么针对普通表的可用回滚段数量就是该值减去32。

23.7.2 配置undo表空间

默认情况下，针对普通表设立的回滚段（第 0 号以及第 33~127 号回滚段）都是被分配到系统表空间的。其中的第 0 号回滚段是一直在系统表空间的，但是第 33~127 号回滚段可以通过配置放到自定义的 undo表空间 中。但是这种配置只能在系统初始化（创建数据目录时）的时候使用，一旦初始化完成，之后就不能再次更改了。我们看一下相关启动参数：

- 通过 innodb_undo_directory 指定 undo表空间 所在的目录，如果没有指定该参数，则默认 undo表空间 所在的目录就是数据目录。
- 通过 innodb_undo_tablespaces 定义 undo表空间 的数量。该参数的默认值为 0 ，表明不创建任何 undo表空间 。

  第 33~127 号回滚段可以平均分布到不同的 undo表空间 中。

  > 小贴士：
  > 如果我们在系统初始化的时候指定了创建了undo表空间，那么系统表空间中的第0号回滚段将处于不可用状态。

比如我们在系统初始化时指定的 innodb_rollback_segments 为 35 ， innodb_undo_tablespaces 为 2 ，这样就会将第 33 、 34 号回滚段分别分布到一个 undo表空间 中。

设立 undo表空间 的一个好处就是在 undo表空间 中的文件大到一定程度时，可以自动的将该 undo表空间 截断（truncate）成一个小文件。而系统表空间的大小只能不断的增大，却不能截断。

24 第24章 一条记录的多幅面孔-事务的隔离级别与MVCC

标签： MySQL是怎样运行的

24.1 事前准备

为了故事的顺利发展，我们需要创建一个表：

```
CREATE TABLE hero (
    number INT,
    name VARCHAR(100),
    country varchar(100),
    PRIMARY KEY (number)
) Engine=InnoDB CHARSET=utf8;
```

小贴士：
注意我们把这个hero表的主键命名为number，而不是id，主要是想和后边要用到的事务id做区别，大家不用大惊小怪哈～

然后向这个表里插入一条数据：

```
INSERT INTO hero VALUES(1, '刘备', '蜀');
```

现在表里的数据就是这样的：

```
mysql> SELECT * FROM hero;
+--------+--------+---------+
| number | name   | country |
+--------+--------+---------+
|      1 | 刘备   | 蜀      |
+--------+--------+---------+
1 row in set (0.00 sec)
```

24.2 事务隔离级别

我们知道 MySQL 是一个 客户端／服务器 架构的软件，对于同一个服务器来说，可以有若干个客户端与之连接，每个客户端与服务器连接上之后，就可以称之为一个会话（ Session ）。每个客户端都可以在自己的会话中向服务器发出请求语句，一个请求语句可能是某个事务的一部分，也就是对于服务器来说可能同时处理多个事务。在事务简介的章节中我们说过事务有一个称之为 隔离性 的特性，理论上在某个事务对某个数据进行访问时，其他事务应该进行排队，当该事务提交之后，其他事务才可以继续访问这个数据。但是这样子的话对性能影响太大，我们既想保持事务的 隔离性 ，又想让服务器在处理访问同一数据的多个事务时性能尽量高些，鱼和熊掌不可得兼，舍一部分 隔离性 而取性能者也。

24.2.1 事务并发执行遇到的问题

怎么个舍弃法呢？我们先得看一下访问相同数据的事务在不保证串行执行（也就是执行完一个再执行另一个）的情况下可能会出现哪些问题：

- 脏写（ Dirty Write ）

  如果一个事务修改了另一个未提交事务修改过的数据，那就意味着发生了 脏写 ，示意图如下：

脏写示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	UPDATE hero SET name ='张飞' WHERE number = 1;	
⑤	COMMIT;	
⑥		ROLLBACK;

如上图， Session A 和 Session B 各开启了一个事务， Session B 中的事务先将 number 列为 1 的记录的 name 列更新为 '关羽' ，然后 Session A 中的事务接着又把这条 number 列为 1 的记录的 name 列更新为 张飞 。如果之后 Session B 中的事务进行了回滚，那么 Session A 中的更新也将不复存在，这种现象就称之为 脏写 。这时 Session A 中的事务就很懵逼，我明明把数据更新了，最后也提交事务了，怎么到最后说自己啥也没干呢？

- 脏读 （ Dirty Read ）

如果一个事务读到了另一个未提交事务修改过的数据，那就意味着发生了 脏读 ，示意图如下：

脏读示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②		BEGIN;
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	SELECT * FROM hero WHERE number = 1; （如果读到列name的值为'关羽'，则意味着发生了脏读）	
⑤	COMMIT;	
⑥		ROLLBACK;

如上图， Session A 和 Session B 各开启了一个事务， Session B 中的事务先将 number 列为 1 的记录的 name 列更新为 '关羽' ，然后 Session A 中的事务再去查询这条 number 为 1 的记录，如果du到列 name 的值为 '关羽' ，而 Session B 中的事务稍后进行了回滚，那么 Session A 中的事务相当于读到了一个不存在的数据，这种现象就称之为 脏读 。

- 不可重复读（Non-Repeatable Read）

如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那就意味着发生了 不可重复读 ，示意图如下：

不可重复读示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM hero WHERE number = 1; (此时读到的列name的值为'刘备')	
③		UPDATE hero SET name ='关羽' WHERE number = 1;
④	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'关羽'，则意味着发生了不可重复读)	
⑤		UPDATE hero SET name ='张飞' WHERE number = 1;
⑥	SELECT * FROM hero WHERE number = 1; (如果读到列name的值为'张飞'，则意味着发生了不可重复读)	

如上图，我们在 Session B 中提交了几个隐式事务（注意是隐式事务，意味着语句结束事务就提交了），这些事务都修改了 number 列为 1 的记录的列 name 的值，每次事务提交之后，如果 Session A 中的事务都可以查看到最新的值，这种现象也被称之为 不可重复读 。

- 幻读（Phantom）

如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，那就意味着发生了 幻读 ，示意图如下：

幻读示意图

发生时间编号	Session A	Session B
①	BEGIN;	
②	SELECT * FROM hero WHERE number > 0; (此时读到的列name的值为'刘备')	
③		INSERT INTO hero VALUES(2, '曹操', '魏');
④	SELECT * FROM hero WHERE number > 0; (如果读到列name的值为'刘备'、'曹操'的记录，则意味着发生了幻读)	

如上图， Session A 中的事务先根据条件 number > 0 这个条件查询表 hero ，得到了 name 列值为 '刘备' 的记录；之后 Session B 中提交了一个隐式事务，该事务向表 hero 中插入了一条新记录；之后 Session A 中的事务再根据相同的条件 number > 0 查询表 hero ，得到的结果集中包含 Session B 中的事务新插入的那条记录，这种现象也被称之为 幻读 。

有的同学会有疑问，那如果 Session B 中是删除了一些符合 number > 0 的记录而不是插入新记录，那 Session A 中之后再根据 number > 0 的条件读取的记录变少了，这种现象算不算 幻读 呢？明确说一下，这种现象不属于 幻读 ， 幻读 强调的是一个事务按照某个相同条件多次读取记录时，后读取时读到了之前没有读到的记录。

> 小贴士：
> 那对于先前已经读到的记录，之后又读取不到这种情况，算啥呢？其实这相当于对每一条记录都发生了不可重复读的现象。幻读只是重点强调了读取到了之前读取没有获取到的记录。

24.2.2 SQL标准中的四种隔离级别

我们上边介绍了几种并发事务执行过程中可能遇到的一些问题，这些问题也有轻重缓急之分，我们给这些问题按照严重性来排一下序：

```
脏写 > 脏读 > 不可重复读 > 幻读
```

我们上边所说的舍弃一部分隔离性来换取一部分性能在这里就体现在：设立一些隔离级别，隔离级别越低，越严重的问题就越可能发生。有一帮人（并不是设计 MySQL 的大叔们）制定了一个所谓的 SQL标准 ，在标准中设立了4个 隔离级别 ：

- READ UNCOMMITTED ：未提交读。
- READ COMMITTED ：已提交读。
- REPEATABLE READ ：可重复读。
- SERIALIZABLE ：可串行化。

SQL标准 中规定，针对不同的隔离级别，并发事务可以发生不同严重程度的问题，具体情况如下：

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

也就是说：

- READ UNCOMMITTED 隔离级别下，可能发生 脏读 、 不可重复读 和 幻读 问题。
- READ COMMITTED 隔离级别下，可能发生 不可重复读 和 幻读 问题，但是不可以发生 脏读 问题。
- REPEATABLE READ 隔离级别下，可能发生 幻读 问题，但是不可以发生 脏读 和 不可重复读 的问题。
- SERIALIZABLE 隔离级别下，各种问题都不可以发生。

脏写 是怎么回事儿？怎么里边都没写呢？这是因为脏写这个问题太严重了，不论是哪种隔离级别，都不允许脏写的情况发生。

24.2.3 MySQL中支持的四种隔离级别

不同的数据库厂商对 SQL标准 中规定的四种隔离级别支持不一样，比方说 Oracle 就只支持 READ COMMITTED 和 SERIALIZABLE 隔离级别。本书中所讨论的 MySQL 虽然支持4种隔离级别，但与 SQL标准 中所规定的各级隔离级别允许发生的问题却有些出入，MySQL在REPEATABLE READ隔离级别下，是可以禁止幻读问题的发生的（关于如何禁止我们之后会详细说明的）。

MySQL 的默认隔离级别为 REPEATABLE READ ，我们可以手动修改一下事务的隔离级别。

24.2.3.1 如何设置事务的隔离级别

我们可以通过下边的语句修改事务的隔离级别：

```
SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL level;
```

其中的 level 可选值有4个：

```
level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
}
```

设置事务的隔离级别的语句中，在 SET 关键字后可以放置 GLOBAL 关键字、 SESSION 关键字或者什么都不放，这样会对不同范围的事务产生不同的影响，具体如下：

- 使用 GLOBAL 关键字（在全局范围影响）：

  比方说这样：

  ```
  SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  ```

  则：
  - 只对执行完该语句之后产生的会话起作用。
  - 当前已经存在的会话无效。
- 使用 SESSION 关键字（在会话范围影响）：

  比方说这样：

  ```
  SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  ```

  则：
  - 对当前会话的所有后续的事务有效
  - 该语句可以在已经开启的事务中间执行，但不会影响当前正在执行的事务。
  - 如果在事务之间执行，则对后续的事务有效。
- 上述两个关键字都不用（只对执行语句后的下一个事务产生影响）：

  比方说这样：

  ```
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  ```

  则：
  - 只对当前会话中下一个即将开启的事务有效。
  - 下一个事务执行完后，后续事务将恢复到之前的隔离级别。
  - 该语句不能在已经开启的事务中间执行，会报错的。

如果我们在服务器启动时想改变事务的默认隔离级别，可以修改启动参数 transaction-isolation 的值，比方说我们在启动服务器时指定了 --transaction-isolation=SERIALIZABLE ，那么事务的默认隔离级别就从原来的 REPEATABLE READ 变成了 SERIALIZABLE 。

想要查看当前会话默认的隔离级别可以通过查看系统变量 transaction_isolation 的值来确定：

```
mysql> SHOW VARIABLES LIKE 'transaction_isolation';
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.02 sec)
```

或者使用更简便的写法：

```
mysql> SELECT @@transaction_isolation;
+-------------------------+
| @@transaction_isolation |
+-------------------------+
| REPEATABLE-READ         |
+-------------------------+
1 row in set (0.00 sec)
```

小贴士：
我们也可以使用设置系统变量transaction_isolation的方式来设置事务的隔离级别，不过我们前边介绍过，一般系统变量只有GLOBAL和SESSION两个作用范围，而这个transaction_isolation却有3个（与上边 SET TRANSACTION ISOLATION LEVEL的语法相对应），设置语法上有些特殊，更多详情可以参见文档：https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html#sysvar_transaction_isolation。
另外，transaction_isolation是在MySQL 5.7.20的版本中引入来替换tx_isolation的，如果你使用的是之前版本的MySQL，请将上述用到系统变量transaction_isolation的地方替换为tx_isolation。

24.3 MVCC原理

24.3.1 版本链

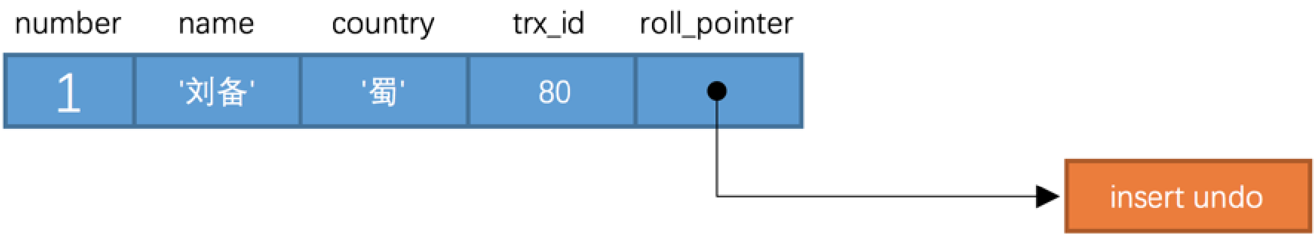
我们前边说过，对于使用 InnoDB 存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列（ row_id 并不是必要的，我们创建的表中有主键或者非NULL的UNIQUE键时都不会包含 row_id 列）：

- trx_id ：每次一个事务对某条聚簇索引记录进行改动时，都会把该事务的 事务id 赋值给 trx_id 隐藏列。
- roll_pointer ：每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到 undo日志 中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

比方说我们的表 hero 现在只包含一条记录：

```
mysql> SELECT * FROM hero;
+--------+--------+---------+
| number | name   | country |
+--------+--------+---------+
|      1 | 刘备   | 蜀      |
+--------+--------+---------+
1 row in set (0.07 sec)
```

假设插入该记录的 事务id 为 80 ，那么此刻该条记录的示意图如下所示：



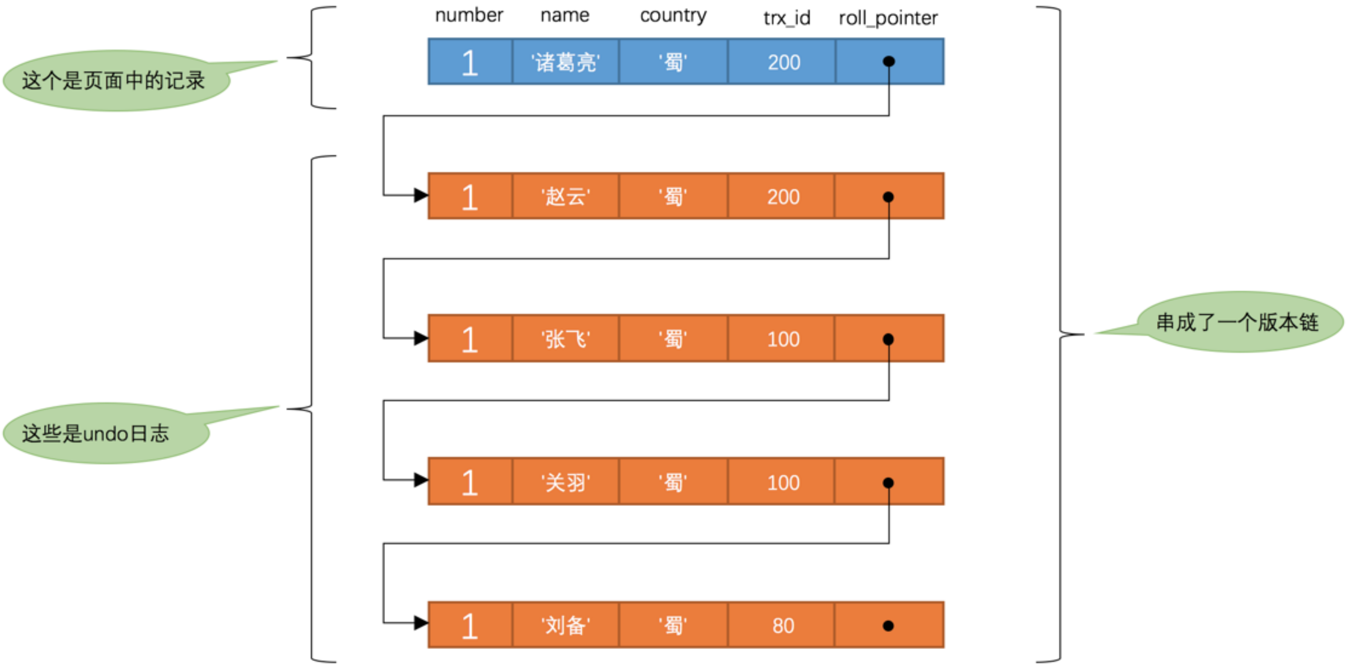
小贴士：
实际上insert undo只在事务回滚时起作用，当事务提交后，该类型的undo日志就没用了，它占用的Undo Log Segment也会被系统回收（也就是该undo日志占用的Undo页面链表要么被重用，要么被释放）。虽然真正的insert undo日志占用的存储空间被释放了，但是roll_pointer的值并不会被清除，roll_pointer属性占用7个字节，第一个比特位就标记着它指向的undo日志的类型，如果该比特位的值为1时，就代表着它zhi向的undo日志类型为insert undo。所以我们之后在画图时都会把insert undo给去掉，大家留意一下就好了。

假设之后两个 事务id 分别为 100 、 200 的事务对这条记录进行 UPDATE 操作，操作流程如下：

发生时间编号	trx 100	trx 200
①	BEGIN;	
②		BEGIN;
③	UPDATE hero SET name = '关羽' WHERE number = 1;	
④	UPDATE hero SET name = '张飞' WHERE number = 1;	
⑤	COMMIT;	
⑥		UPDATE hero SET name = '赵云' WHERE number = 1;
⑦		UPDATE hero SET name = '诸葛亮' WHERE number = 1;
⑧		COMMIT;

小贴士：
能不能在两个事务中交叉更新同一条记录呢？哈哈，这不就是一个事务修改了另一个未提交事务修改过的数据，沦为了脏写了么？InnoDB使用锁来保证不会有脏写情况的发生，也就是在第一个事务更新了某条记录后，就会给这条记录加锁，另一个事务再次更新时就需要等待第一个事务提交了，把锁释放之后才可以继续更新。关于锁的更多细节我们后续的文章中再唠叨哈～

每次对记录进行改动，都会记录一条 undo日志 ，每条 undo日志 也都有一个 roll_pointer 属性（ INSERT 操作对应的 undo日志 没有该属性，因为该记录并没有更早的版本），可以将这些 undo日志 都连起来，串成一个链表，所以现在的情况就像下图一样：



对该记录每次更新后，都会将旧值放到一条 undo日志 中，就算是该记录的一个旧版本，随着更新次数的增多，所有的版本都会被 roll_pointer 属性连接成一个链表，我们把这个链表称之为 版本链 ，版本链的头节点就是当前记录最新的值。另外，每个版本中还包含生成该版本时对应的 事务id ，这个信息很重要，我们稍后就会用到。

24.3.2 ReadView

对于使用 READ UNCOMMITTED 隔离级别的事务来说，由于可以读到未提交事务修改过的记录，所以直接读取记录的最新版本就好了；对于使用 SERIALIZABLE 隔离级别的事务来说，设计 InnoDB 的大叔规定使用加锁的方式来访问记录（加锁是啥我们后续文章中说哈）；对于使用 READ COMMITTED 和 REPEATABLE READ 隔离级别的事务来说，都必须保证读到已经提交了的事务修改过的记录，也就是说假如另一个事务已经修改了记录但是尚未提交，是不能直接读取最新版本的记录的，核心问题就是：需要判断一下版本链中的哪个版本是当前事务可见的。为此，设计 InnoDB 的大叔提出了一个 ReadView 的概念，这个 ReadView 中主要包含4个比较重要的内容：

- m_ids ：表示在生成 ReadView 时当前系统中活跃的读写事务的 事务id 列表。
- min_trx_id ：表示在生成 ReadView 时当前系统中活跃的读写事务中最小的 事务id ，也就是 m_ids 中的最小值。
- max_trx_id ：表示生成 ReadView 时系统中应该分配给下一个事务的 id 值。

> 小贴士：
> 注意max_trx_id并不是m_ids中的最大值，事务id是递增分配的。比方说现在有id为1，2，3这三个事务，之后id为3的事务提交了。那么一个新的读事务在生成ReadView时，m_ids就包括1和2，min_trx_id的值就是1，max_trx_id的值就是4。

- creator_trx_id ：表示生成该 ReadView 的事务的 事务id 。

> 小贴士：
> 我们前边说过，只有在对表中的记录做改动时（执行INSERT、DELETE、UPDATE这些语句时）才会为事务分配事务id，否则在一个只读事务中的事务id值都默认为0。

有了这个 ReadView ，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

- 如果被访问版本的 trx_id 属性值与 ReadView 中的 creator_trx_id 值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。

- 如果被访问版本的 trx_id 属性值小于 ReadView 中的 min_trx_id 值，表明生成该版本的事务在当前事务生成 ReadView 前已经提交，所以该版本可以被当前事务访问。
- 如果被访问版本的 trx_id 属性值大于 ReadView 中的 max_trx_id 值，表明生成该版本的事务在当前事务生成 ReadView 后才开启，所以该版本不可以被当前事务访问。
- 如果被访问版本的 trx_id 属性值在 ReadView 的 min_trx_id 和 max_trx_id 之间，那就需要判断一下 trx_id 属性值是不是在 m_ids 列表中，如果在，说明创建 ReadView 时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建 ReadView 时生成该版本的事务已经被提交，该版本可以被访问。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本。如果最后一个版本也不可见的话，那么就意味着该条记录对该事务完全不可见，查询结果就不包含该记录。

在 MySQL 中， READ COMMITTED 和 REPEATABLE READ 隔离级别的的一个非常大的区别就是它们生成ReadView的时机不同。我们还是以表 hero 为例来，假设现在表 hero 中只有一条由 事务id 为 80 的事务插入的一条记录：

```
mysql> SELECT * FROM hero;
+--------+--------+---------+
| number | name   | country |
+--------+--------+---------+
|      1 | 刘备   | 蜀      |
+--------+--------+---------+
1 row in set (0.07 sec)
```

接下来看一下 READ COMMITTED 和 REPEATABLE READ 所谓的生成ReadView的时机不同到底不同在哪里。

24.3.2.1 READ COMMITTED —— 每次读取数据前都生成一个ReadView

比方说现在系统里有两个 事务id 分别为 100 、 200 的事务在执行：

```
# Transaction 100
BEGIN;

UPDATE hero SET name = '关羽' WHERE number = 1;

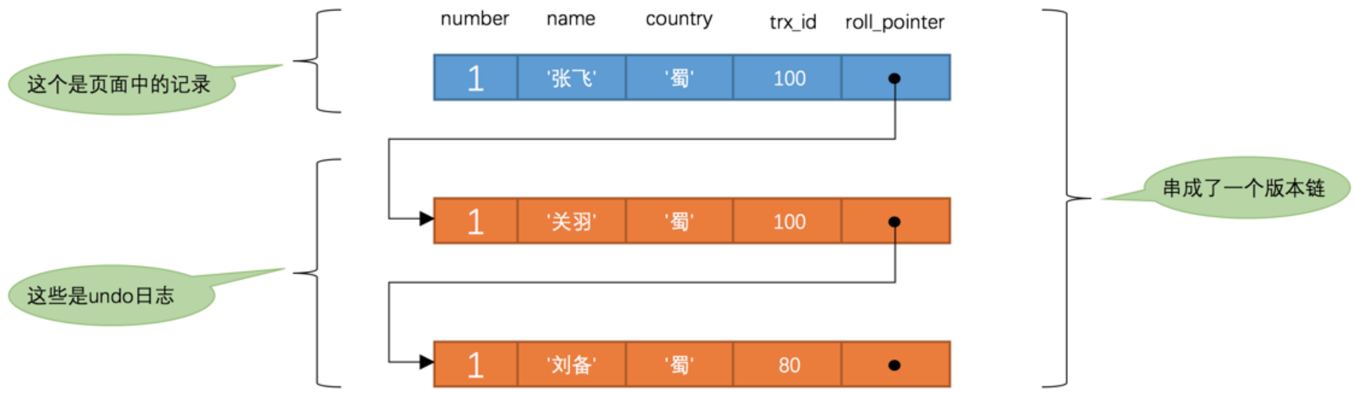
UPDATE hero SET name = '张飞' WHERE number = 1;

# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

```
小贴士：
再次强调一遍，事务执行过程中，只有在第一次真正修改记录时（比如使用INSERT、DELETE、UPDATE语句），才会被分配一个单独的事务id，这个事务id是递增的。所以我们才在Transaction 200中更新一些别的表的记录，目的是让它分配事务id。
```

此刻，表 hero 中 number 为 1 的记录得到的版本链表如下所示：



假设现在有一个使用 READ COMMITTED 隔离级别的事务开始执行：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值为'刘备'
```

这个 SELECT1 的执行过程如下：

- 在执行 SELECT 语句时会先生成一个 ReadView ， ReadView 的 m_ids 列表的内容就是 [100, 200] ， min_trx_id 为 100 ， max_trx_id 为 201 ， creator_trx_id 为 0 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 name 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 name 的内容是 '关羽' ，该版本的 trx_id 值也为 100 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 name 的内容是 '刘备' ，该版本的 trx_id 值为 80 ，小于 ReadView 中的 min_trx_id 值 100 ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 name 为 '刘备' 的记录。

之后，我们把 事务id 为 100 的事务提交一下，就像这样：

```
# Transaction 100
BEGIN;

UPDATE hero SET name = '关羽' WHERE number = 1;

UPDATE hero SET name = '张飞' WHERE number = 1;

COMMIT;
```

然后再到 事务id 为 200 的事务中更新一下表 hero 中 number 为 1 的记录：

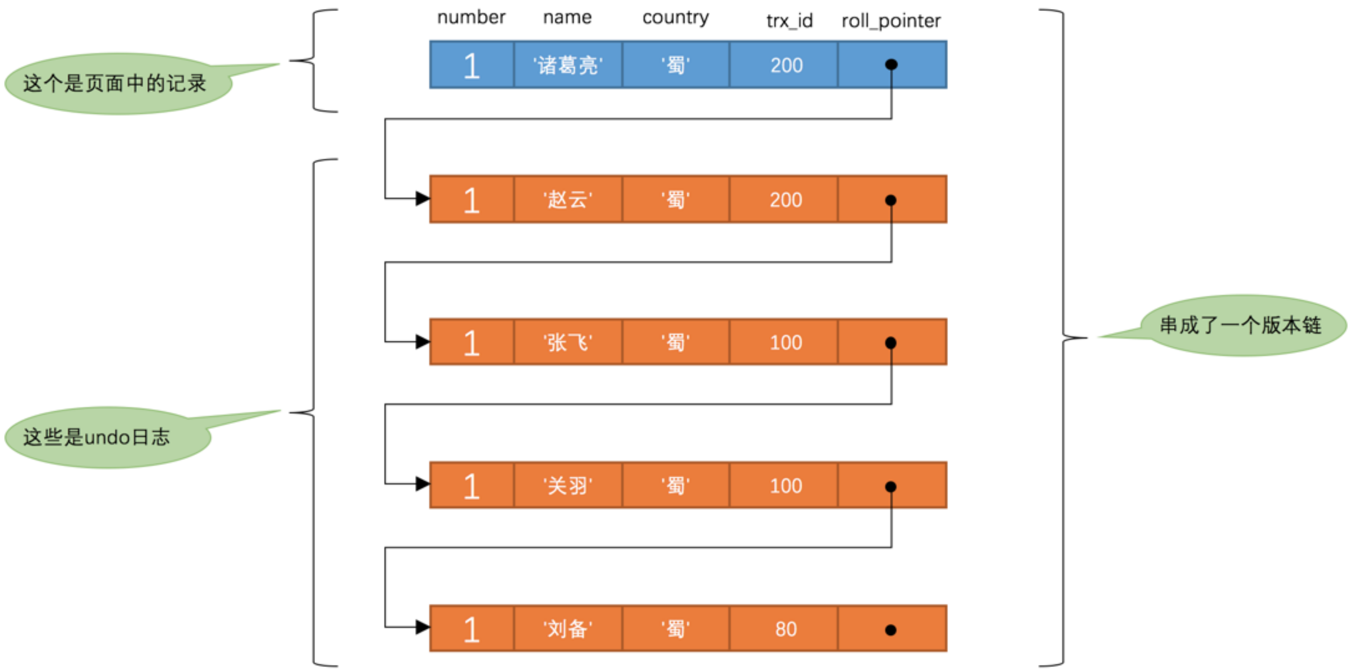
```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE hero SET name = '赵云' WHERE number = 1;

UPDATE hero SET name = '诸葛亮' WHERE number = 1;
```

此刻，表 hero 中 number 为 1 的记录的版本链就长这样：



然后再到刚才使用 READ COMMITTED 隔离级别的事务中继续查找这个 number 为 1 的记录，如下：

```
# 使用READ COMMITTED隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200均未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值为'张飞'
```

这个 SELECT2 的执行过程如下：

- 在执行 SELECT 语句时会又会单独生成一个 ReadView ，该 ReadView 的 m_ids 列表的内容就是 [200] （ 事务id 为 100 的那个事务已经提交了，所以再次生成快照时就没有它了）， min_trx_id 为 200 ， max_trx_id 为 201 ， creator_trx_id 为 0 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 name 的内容是 '诸葛亮' ，该版本的 trx_id 值为 200 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 name 的内容是 '赵云' ，该版本的 trx_id 值为 200 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 name 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，小于 ReadView 中的 min_trx_id 值 200 ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 name 为 '张飞' 的记录。

以此类推，如果之后 事务id 为 200 的记录也提交了，再此在使用 READ COMMITTED 隔离级别的事务中查询表 hero 中 number 值为 1 的记录时，得到的结果就是 '诸葛亮' 了，具体流程我们就不分析了。总结一下就是：使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成一个独立的ReadView。

24.3.2.2 REPEATABLE READ —— 在第一次读取数据时生成一个ReadView

对于使用 REPEATABLE READ 隔离级别的事务来说，只会在第一次执行查询语句时生成一个 ReadView ，之后的查询就不会重复生成了。我们还是用例子看一下是什么效果。

比方说现在系统里有两个 事务id 分别为 100 、 200 的事务在执行：

```
# Transaction 100
BEGIN;

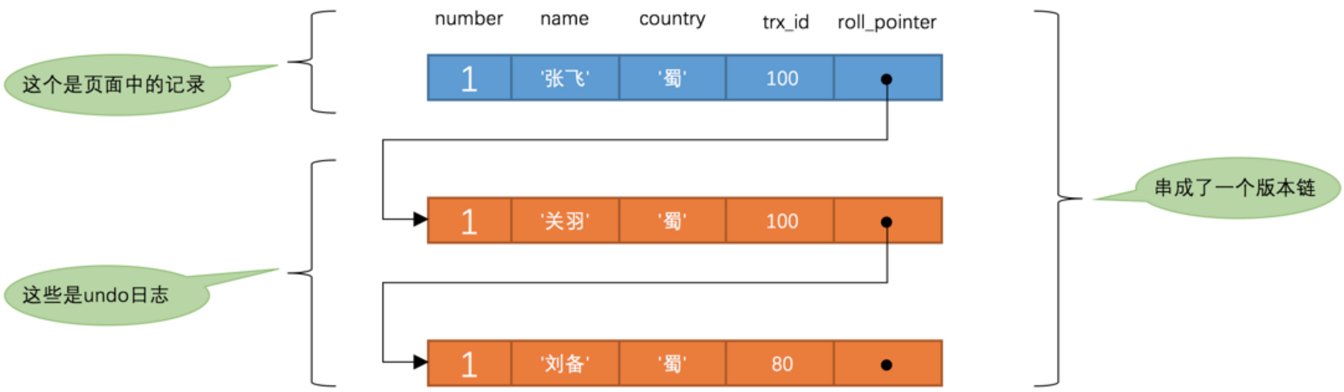
UPDATE hero SET name = '关羽' WHERE number = 1;

UPDATE hero SET name = '张飞' WHERE number = 1;

# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...
```

此刻，表 hero 中 number 为 1 的记录得到的版本链表如下所示：



假设现在有一个使用 REPEATABLE READ 隔离级别的事务开始执行：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值为'刘备'
```

这个 SELECT1 的执行过程如下：

- 在执行 SELECT 语句时会先生成一个 ReadView ， ReadView 的 m_ids 列表的内容就是 [100, 200] ， min_trx_id 为 100 ， max_trx_id 为 201 ， creator_trx_id 为 0 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 name 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 name 的内容是 '关羽' ，该版本的 trx_id 值也为 100 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 name 的内容是 '刘备' ，该版本的 trx_id 值为 80 ，小于 ReadView 中的 min_trx_id 值 100 ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 name 为 '刘备' 的记录。

之后，我们把 事务id 为 100 的事务提交一下，就像这样：

```
# Transaction 100
BEGIN;

UPDATE hero SET name = '关羽' WHERE number = 1;

UPDATE hero SET name = '张飞' WHERE number = 1;

COMMIT;
```

然后再到 事务id 为 200 的事务中更新一下表 hero 中 number 为 1 的记录：

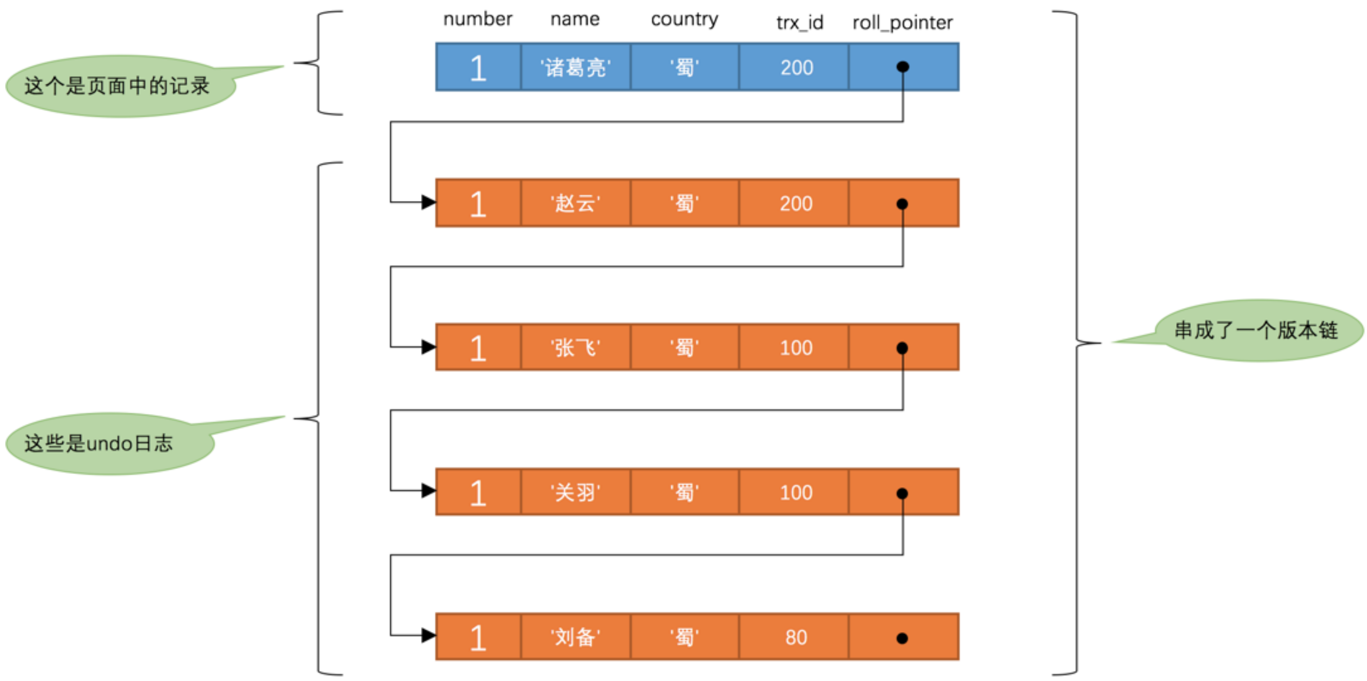
```
# Transaction 200
BEGIN;

# 更新了一些别的表的记录
...

UPDATE hero SET name = '赵云' WHERE number = 1;

UPDATE hero SET name = '诸葛亮' WHERE number = 1;
```

此刻，表 hero 中 number 为 1 的记录的版本链就长这样：



然后再到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个 number 为 1 的记录，如下：

```
# 使用REPEATABLE READ隔离级别的事务
BEGIN;

# SELECT1：Transaction 100、200均未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值为'刘备'

# SELECT2：Transaction 100提交，Transaction 200未提交
SELECT * FROM hero WHERE number = 1; # 得到的列name的值仍为'刘备'
```

这个 SELECT2 的执行过程如下：

- 因为当前事务的隔离级别为 REPEATABLE READ ，而之前在执行 SELECT1 时已经生成过 ReadView 了，所以此时直接复用之前的 ReadView ，之前的 ReadView 的 m_ids 列表的内容就是 [100, 200] ， min_trx_id 为 100 ， max_trx_id 为 201 ， creator_trx_id 为 0 。
- 然后从版本链中挑选可见的记录，从图中可以看出，最新版本的列 name 的内容是 '诸葛亮' ，该版本的 trx_id 值为 200 ，在 m_ids 列表内，所以不符合可见性要求，根据 roll_pointer 跳到下一个版本。
- 下一个版本的列 name 的内容是 '赵云' ，该版本的 trx_id 值为 200 ，也在 m_ids 列表内，所以也不符合要求，继续跳到下一个版本。
- 下一个版本的列 name 的内容是 '张飞' ，该版本的 trx_id 值为 100 ，而 m_ids 列表中是包含值为 100 的 事务id 的，所以该版本也不符合要求，同理下一个列 name 的内容是 '关羽' 的版本也不符合要求。继续跳到下一个版本。
- 下一个版本的列 name 的内容是 '刘备' ，该版本的 trx_id 值为 80 ，小于 ReadView 中的 min_trx_id 值 100 ，所以这个版本是符合要求的，最后返回给用户的版本就是这条列 c 为 '刘备' 的记录。

也就是说两次 SELECT 查询得到的结果是重复的，记录的列 c 值都是 '刘备' ，这就是 可重复读 的含义。如果我们之后再把 事务id 为 200 的记录提交了，然后再到刚才使用 REPEATABLE READ 隔离级别的事务中继续查找这个 number 为 1 的记录，得到的结果还是 '刘备' ，具体执行过程大家可以自己分析一下。

24.3.3 MVCC小结

从上边的描述中我们可以看出来，所谓的 MVCC （Multi-Version Concurrency Control ，多版本并发控制）指的就是在使用 READ COMMITTD 、 REPEATABLE READ 这两种隔离级别的事务在执行普通的 SEELCT 操作时访问记录的版本链的过程，这样子可以使不同事务的 读-写 、 写-读 操作并发执行，从而提升系统性能。 READ COMMITTD 、 REPEATABLE READ 这两个隔离级别的一个很大不同就是：生成ReadView的时机不同，READ COMMITTD在每一次进行普通SELECT操作前都会生成一个ReadView，而REPEATABLE READ只在第一次进行普通SELECT操作前生成一个ReadView，之后的查询操作都重复使用这个ReadView就好了。

> 小贴士：
> 我们之前说执行DELETE语句或者更新主键的UPDATE语句并不会立即把对应的记录完全从页面中删除，而是执行一个所谓的delete mark操作，相当于只是对记录打上了一个删除标志位，这主要就是为MVCC服务的，大家可以对比上边举的例子自己试想一下怎么使用。
> 另外，所谓的MVCC只是在我们进行普通的SEELCT查询时才生效，截止到目前我们所见的所有SELECT语句都算是普通的查询，至于啥是个不普通的查询，我们稍后再说哈～

24.4 关于purge

大家有没有发现两件事儿：

- 我们说 insert undo 在事务提交之后就可以被释放掉了，而 update undo 由于还需要支持 MVCC ，不能立即删除掉。
- 为了支持 MVCC ，对于 delete mark 操作来说，仅仅是在记录上打一个删除标记，并没有真正将它删除掉。

随着系统的运行，在确定系统中包含最早产生的那个 ReadView 的事务不会再访问某些 update undo日志 以及被打了删除标记的记录后，有一个后台运行的 purge线程 会把它们真正的删除掉。关于更多的purge细节，我们将放到纸质书中进行详细唠叨，不见不散哈～

25 第25章 工作面试老大难-锁

标签： MySQL是怎样运行的

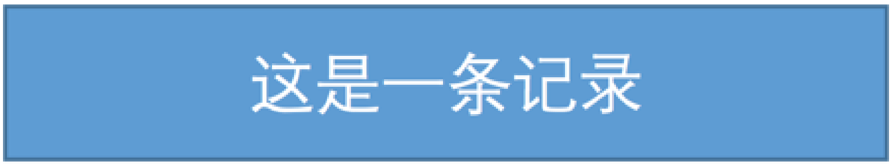
25.1 解决并发事务带来问题的两种基本方式

上一章唠叨了事务并发执行时可能带来的各种问题，并发事务访问相同记录的情况大致可以划分为3种：

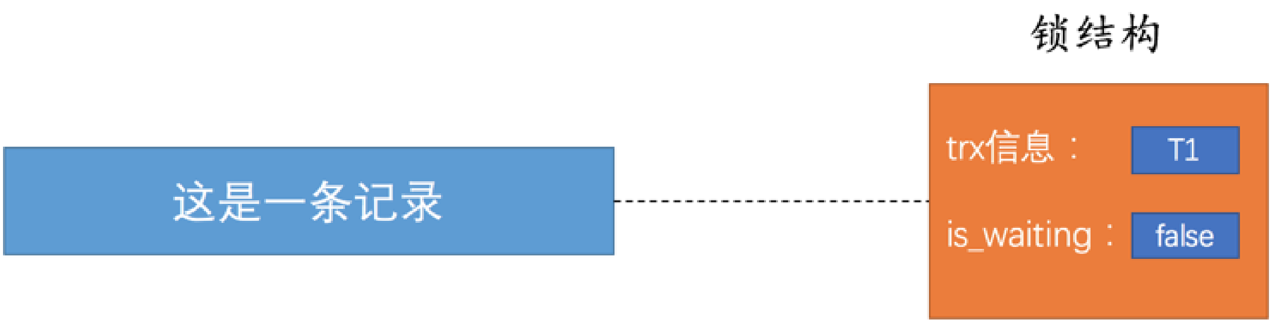
- 读-读 情况：即并发事务相继读取相同的记录。

  读取操作本身不会对记录有一毛钱影响，并不会引起什么问题，所以允许这种情况的发生。
- 写-写 情况：即并发事务相继对相同的记录做出改动。

  我们前边说过，在这种情况下会发生 脏写 的问题，任何一种隔离级别都不允许这种问题的发生。所以在多个未提交事务相继对一条记录做改动时，需要让它们排队执行，这个排队的过程其实是通过 锁 来实现的。这个所谓的 锁 其实是一个内存中的结构，在事务执行前本来是没有锁的，也就是说一开始是没有 锁结构 和记录进行关联的，如图所示：



  当一个事务想对这条记录做改动时，首先会看看内存中有没有与这条记录关联的 锁结构 ，当没有的时候就会在内存中生成一个 锁结构 与之关联。比方说事务 T1 要对这条记录做改动，就需要生成一个 锁结构 与之关联：

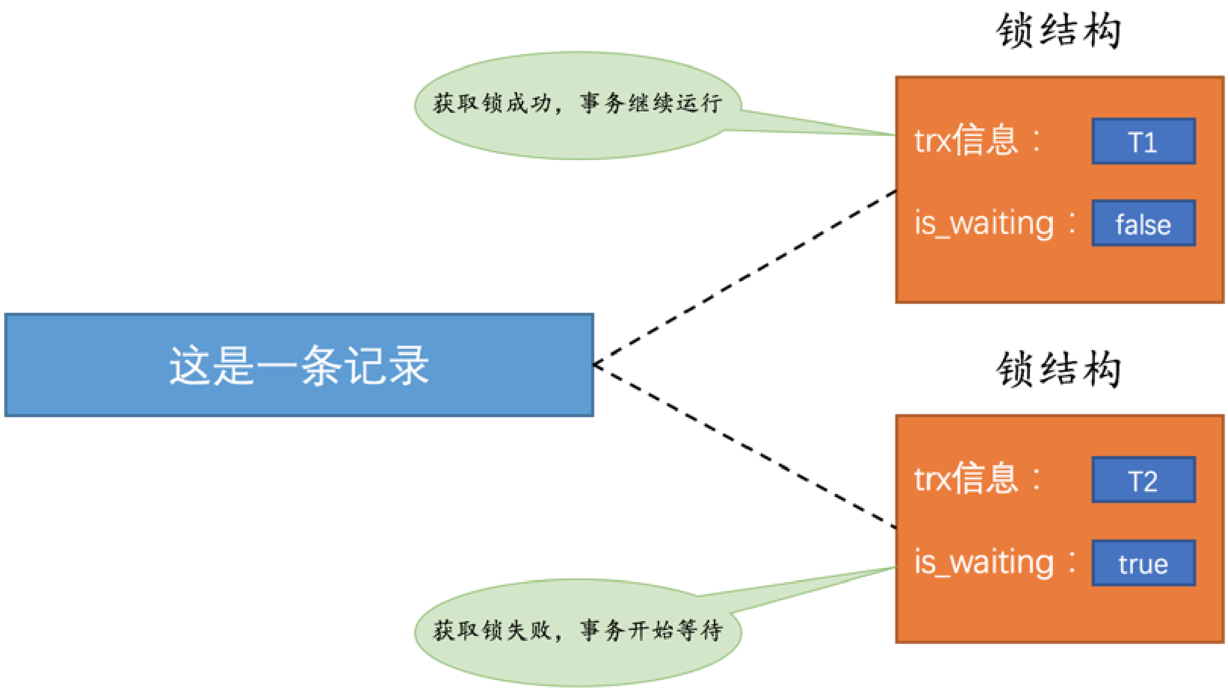


  其实在 锁结构 里有很多信息，不过为了简化理解，我们现在只把两个比较重要的属性拿了出来：
  - trx信息 ：代表这个锁结构是哪个事务生成的。
  - is_waiting ：代表当前事务是否在等待。

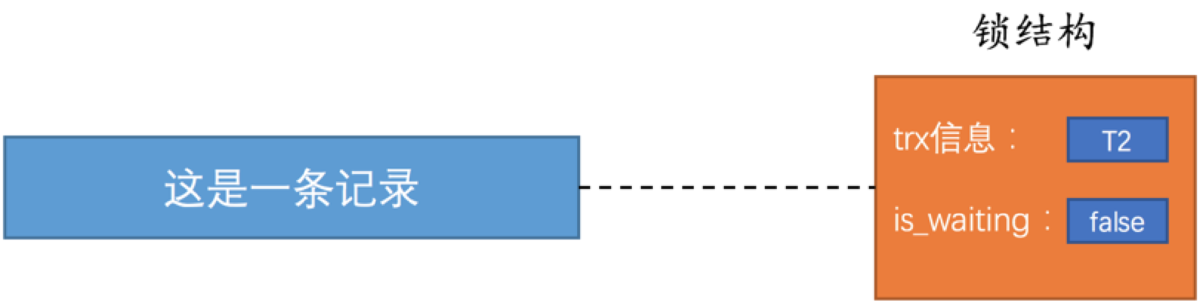
  如图所示，当事务 T1 改动了这条记录后，就生成了一个 锁结构 与该记录关联，因为之前没有别的事务为这条记录加锁，所以 is_waiting 属性就是 false ，我们把这个场景就称之为获取锁成功，或者加锁成功，然后就可以继续执行操作了。

  在事务 T1 提交之前，另一个事务 T2 也想对该记录做改动，那么先去看看有没有 锁结构 与这条记录关联，发现有一个 锁结构 与之关联后，然后也生成了一个 锁结构 与这条记录关联，不过 锁结构 的 is_waiting 属性值为 true ，表示当前事务需要等待，我们把这个场景就称之为获取锁失败，或者加锁

失败，或者没有成功的获取到锁，画个图表示就是这样：



在事务 T1 提交之后，就会把该事务生成的 锁结构 释放掉，然后看看还有没有别的事务在等待获取锁，发现了事务 T2 还在等待获取锁，所以把事务 T2 对应的锁结构的 is_waiting 属性设置为 false ，然后把该事务对应的线程唤醒，让它继续执行，此时事务 T2 就算获取到锁了。效果图就是这样：



我们总结一下后续内容中可能用到的几种说法，以免大家混淆：

- 不加锁

  意思就是不需要在内存中生成对应的 锁结构 ，可以直接执行操作。
- 获取锁成功，或者加锁成功

  意思就是在内存中生成了对应的 锁结构 ，而且锁结构的 is_waiting 属性为 false ，也就是事务可以继续执行操作。
- 获取锁失败，或者加锁失败，或者没有获取到锁

  意思就是在内存中生成了对应的 锁结构 ，不过锁结构的 is_waiting 属性为 true ，也就是事务需要等待，不可以继续执行操作。

> 小贴士：
> 这里只是对锁结构做了一个非常简单的描述，我们后边会详细唠叨唠叨锁结构的，稍安勿躁。

- 读-写 或 写-读 情况：也就是一个事务进行读取操作，另一个进行改动操作。

我们前边说过，这种情况下可能发生 脏读 、 不可重复读 、 幻读 的问题。

> 小贴士：
> 幻读问题的产生是因为某个事务读了一个范围的记录，之后别的事务在该范围内插入了新记录，该事务再次读取该范围的记录时，可以读到新插入的记录，所以幻读问题准确的说并不是因为读取和写入一条相同记录而产生的，这一点要注意一下。

SQL标准 规定不同隔离级别下可能发生的问题不一样：

- 在 READ UNCOMMITTED 隔离级别下， 脏读 、 不可重复读 、 幻读 都可能发生。
- 在 READ COMMITTED 隔离级别下， 不可重复读 、 幻读 可能发生， 脏读 不可以发生。
- 在 REPEATABLE READ 隔离级别下， 幻读 可能发生， 脏读 和 不可重复读 不可以发生。
- 在 SERIALIZABLE 隔离级别下，上述问题都不可以发生。

不过各个数据库厂商对 SQL标准 的支持都可能不一样，与 SQL标准 不同的一点就是， MySQL 在 REPEATABLE READ 隔离级别实际上就已经解决了 幻读 问题。

怎么解决 脏读 、 不可重复读 、 幻读 这些问题呢？其实有两种可选的解决方案：

- 方案一：读操作利用多版本并发控制（ MVCC ），写操作进行 加锁 。

所谓的 MVCC 我们在前一章有过详细的描述，就是通过生成一个 ReadView ，然后通过 ReadView 找到符合条件的记录版本（历史版本是由 undo日志 构建的），其实就像是在生成 ReadView 的那个时刻做了一次时间静止（就像用相机拍了一个快照），查询语句只能读到在生成 ReadView 之前已提交事务所做的更改，在生成 ReadView 之前未提交的事务或者之后才开启的事务所做的更改是看不到的。而写操作肯定针对的是最新版本的记录，读记录的历史版本和改动记录的最新版本本身并不冲突，也就是采用 MVCC 时， 读-写 操作并不冲突。

> 小贴士：
> 我们说过普通的SELECT语句在READ COMMITTED和REPEATABLE READ隔离级别下会使用到MVCC读取记录。在READ COMMITTED隔离级别下，一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView，ReadView的存在本身就保证了事务不可以读取到未提交的事务所做的更改，也就是避免了脏读现象；REPEATABLE READ隔离级别下，一个事务在执行过程中只有第一次执行SELECT操作才会生成一个ReadView，之后的SELECT操作都复用这个ReadView，这样也就避免了不可重复读和幻读的问题。

- 方案二：读、写操作都采用 加锁 的方式。

如果我们的一些业务场景不允许读取记录的旧版本，而是每次都必须去读取记录的最新版本，比方在银行存款的事务中，你需要先把账户的余额读出来，然后将其加上本次存款的数额，最后再写到数据库中。在将账户余额读取出来后，就不想让别的事务再访问该余额，直到本次存款事务执行完成，其他事务才可以访问账户的余额。这样在读取记录的时候也就需要对其进行 加锁 操作，这样也就意味着 读 操作和 写 操作也像 写-写 操作那样排队执行。

> 小贴士：
> 我们说脏读的产生是因为当前事务读取了另一个未提交事务写的一条记录，如果另一个事务在写记录的时候就给这条记录加锁，那么当前事务就无法继续读取该记录了，所以也就不会有脏读问题的产生了。不可重复读的产生是因为当前事务先读取一条记录，另外一个事务对该记录做了改动之后并提交之后，当前事务再次读取时会获得不同的值，如果在当前事务读取记录时就给该记录加锁，那么另一个事务就无法修改该记录，自然也不会发生不可重复读了。我们说幻读问题的产生是因为当前事务读取了一个范围的记录，然后另外的事务向该范围内插入了新记录，当前事务再次读取该范围的记录时发现了新插入的新记录，我们把新插入的那些记录称之为幻影记录。采用加锁的方式解决幻读问题就有那么一丢丢麻烦了，因为当前事务在第一次读取记录时那些幻影记录并不存在，所以读取的时候加锁就有点尴尬 —— 因为你并不知道给谁加锁，没关系，这难不倒设计InnoDB的大叔的，我们稍后揭晓答案，稍安勿躁。

很明显，采用 MVCC 方式的话， 读-写 操作彼此并不冲突，性能更高，采用 加锁 方式的话， 读-写 操作彼此需要排队执行，影响性能。一般情况下我们当然愿意采用 MVCC 来解决 读-写 操作并发执行的问题，但是业务在某些特殊情况下，要求必须采用 加锁 的方式执行，那也是没有办法的事。

25.1.1 一致性读（Consistent Reads）

事务利用 MVCC 进行的读取操作称之为 一致性读 ，或者 一致性无锁读 ，有的地方也称之为 快照读 。所有普通的 SELECT 语句（ plain SELECT ）在 READ COMMITTED 、 REPEATABLE READ 隔离级别下都算是 一致性读 ，比方说：

```
SELECT * FROM t;
SELECT * FROM t1 INNER JOIN t2 ON t1.col1 = t2.col2
```

一致性读 并不会对表中的任何记录做 加锁 操作，其他事务可以自由的对表中的记录做改动。

25.1.2 锁定读（Locking Reads）

25.1.2.1 共享锁和独占锁

我们前边说过，并发事务的 读-读 情况并不会引起什么问题，不过对于 写-写 、 读-写 或 写-读 这些情况可能会引起一些问题，需要使用 MVCC 或者 加锁 的方式来解决它们。在使用 加锁 的方式解决问题时，由于既要允许 读-读 情况不受影响，又要使 写-写 、 读-写 或 写-读 情况中的操作相互阻塞，所以设计 MySQL 的大叔给锁分了个类：

- 共享锁 ，英文名： Shared Locks ，简称 S锁 。在事务要读取一条记录时，需要先获取该记录的 S锁 。
- 独占锁 ，也常称 排他锁 ，英文名： Exclusive Locks ，简称 X锁 。在事务要改动一条记录时，需要先获取该记录的 X锁 。

假如事务 T1 首先获取了一条记录的 S锁 之后，事务 T2 接着也要访问这条记录：

- 如果事务 T2 想要再获取一个记录的 S锁 ，那么事务 T2 也会获得该锁，也就意味着事务 T1 和 T2 在该记录上同时持有 S锁 。
- 如果事务 T2 想要再获取一个记录的 X锁 ，那么此操作会被阻塞，直到事务 T1 提交之后将 S锁 释放掉。

如果事务 T1 首先获取了一条记录的 X锁 之后，那么不管事务 T2 接着想获取该记录的 S锁 还是 X锁 都会被阻塞，直到事务 T1 提交。

所以我们说 S锁 和 S锁 是兼容的， S锁 和 X锁 是不兼容的， X锁 和 X锁 也是不兼容的，画个表表示一下就是这样：

兼容性	X	S
X	不兼容	不兼容
S	不兼容	兼容

25.1.2.2 锁定读的语句

我们前边说在采用 加锁 方式解决 脏读 、 不可重复读 、 幻读 这些问题时，读取一条记录时需要获取一下该记录的 S锁 ，其实这是不严谨的，有时候想在读取记录时就获取记录的 X锁 ，来禁止别的事务读写该记录，为此设计 MySQL 的大叔提出了两种比较特殊的 SELECT 语句格式：

- 对读取的记录加 S锁 ：

  ```
  SELECT ... LOCK IN SHARE MODE;
  ```

也就是在普通的 SELECT 语句后边加 LOCK IN SHARE MODE ，如果当前事务执行了该语句，那么它会为读取到的记录加 S锁 ，这样允许别的事务继续获取这些记录的 S锁 （比方说别的事务也使用 SELECT ... LOCK IN SHARE MODE 语句来读取这些记录），但是不能获取这些记录的 X锁 （比方说使用 SELECT ... FOR UPDATE 语句来读取这些记录，或者直接修改这些记录）。如果别的事务想要获取这些记录的 X锁 ，那么它们会阻塞，直到当前事务提交之后将这些记录上的 S锁 释放掉。

- 对读取的记录加 X锁 ：

```
SELECT ... FOR UPDATE;
```

也就是在普通的 SELECT 语句后边加 FOR UPDATE ，如果当前事务执行了该语句，那么它会为读取到的记录加 X锁 ，这样既不允许别的事务获取这些记录的 S锁 （比方说别的事务使用 SELECT ... LOCK IN SHARE MODE 语句来读取这些记录），也不允许获取这些记录的 X锁 （比方也说使用 SELECT ... FOR UPDATE 语句来读取这些记录，或者直接修改这些记录）。如果别的事务想要获取这些记录的 S锁 或者 X锁 ，那么它们会阻塞，直到当前事务提交之后将这些记录上的 X锁 释放掉。

关于更多 锁定读 的加锁细节我们稍后会详细唠叨，稍安勿躁。

25.1.3 写操作

平常所用到的 写操作 无非是 DELETE 、 UPDATE 、 INSERT 这三种：

- DELETE ：

  对一条记录做 DELETE 操作的过程其实是先在 B+ 树中定位到这条记录的位置，然后获取一下这条记录的 X锁 ，然后再执行 delete mark 操作。我们也可以把这个定位待删除记录在 B+ 树中位置的过程看成是一个获取 X锁 的 锁定读 。

- UPDATE ：

  在对一条记录做 UPDATE 操作时分为三种情况：

  - 如果未修改该记录的键值并且被更新的列占用的存储空间在修改前后未发生变化，则先在 B+ 树中定位到这条记录的位置，然后再获取一下记录的 X锁 ，最后在原记录的位置进行修改操作。其实我们也可以把这个定位待修改记录在 B+ 树中位置的过程看成是一个获取 X锁 的 锁定读 。
  - 如果未修改该记录的键值并且至少有一个被更新的列占用的存储空间在修改前后发生变化，则先在 B+ 树中定位到这条记录的位置，然后获取一下记录的 X锁 ，将该记录彻底删除掉（就是把记录彻底移入垃圾链表），最后再插入一条新记录。这个定位待修改记录在 B+ 树中位置的过程看成是一个获取 X锁 的 锁定读 ，新插入的记录由 INSERT 操作提供的 隐式锁 进行保护。
  - 如果修改了该记录的键值，则相当于在原记录上做 DELETE 操作之后再来一次 INSERT 操作，加锁操作就需要按照 DELETE 和 INSERT 的规则进行了。

- INSERT ：

  一般情况下，新插入一条记录的操作并不加锁，设计 InnoDB 的大叔通过一种称之为 隐式锁 的东东来保护这条新插入的记录在本事务提交前不被别的事务访问，更多细节我们后边看哈~

> 小贴士：
> 当然，在一些特殊情况下INSERT操作也是会获取锁的，具体情况我们后边唠叨。

25.2 多粒度锁

我们前边提到的 锁 都是针对记录的，也可以被称之为 行级锁 或者 行锁 ，对一条记录加锁影响的也只是这条记录而已，我们就说这个锁的粒度比较细；其实一个事务也可以在 表 级别进行加锁，自然就被称之为 表级锁 或者 表锁 ，对一个表加锁影响整个表中的记录，我们就说这个锁的粒度比较粗。给表加的锁也可以分为 共享锁 （ S锁 ）和 独占锁 （ X锁 ）：

- 给表加 S锁 ：

如果一个事务给表加了 `S锁` ，那么：

- 别的事务可以继续获得该表的 `S锁`
- 别的事务可以继续获得该表中的某些记录的 `S锁`
- 别的事务不可以继续获得该表的 `X锁`
- 别的事务不可以继续获得该表中的某些记录的 `X锁`

- 给表加 `X锁` ：

  如果一个事务给表加了 `X锁` （意味着该事务要独占这个表），那么：

  - 别的事务不可以继续获得该表的 `S锁`
  - 别的事务不可以继续获得该表中的某些记录的 `S锁`
  - 别的事务不可以继续获得该表的 `X锁`
  - 别的事务不可以继续获得该表中的某些记录的 `X锁`

上边看着有点啰嗦，为了更好的理解这个表级别的 `S锁` 和 `X锁` ，我们举一个现实生活中的例子。不知道各位同学都上过大学没，我们以大学教学楼中的教室为例来分析一下加锁的情况：

- 教室一般都是公用的，我们可以随便选教室进去上自习。当然，教室不是自家的，一间教室可以容纳很多同学同时上自习，每当一个人进去上自习，就相当于在教室门口挂了一把 `S锁` ，如果很多同学都进去上自习，相当于教室门口挂了很多把 `S锁` （类似行级别的 `S锁` ）。
- 有的时候教室会进行检修，比方说换地板，换天花板，换灯管啥的，这些维修项目并不能同时开展。如果教室针对某个项目进行检修，就不允许别的同学来上自习，也不允许其他维修项目进行，此时相当于教室门口会挂一把 `X锁` （类似行级别的 `X锁` ）。

上边提到的这两种锁都是针对 `教室` 而言的，不过有时候我们会有一些特殊的需求：

- 有领导要来参观教学楼的环境。

  校领导考虑并不想影响同学们上自习，但是此时不能有教室处于维修状态，所以可以在教学楼门口放置一把 `S锁` （类似表级别的 `S锁` ）。此时：

  - 来上自习的学生们看到教学楼门口有 `S锁` ，可以继续进入教学楼上自习。
  - 修理工看到教学楼门口有 `S锁` ，则先在教学楼门口等着，啥时候领导走了，把教学楼的 `S锁` 撤掉再进入教学楼维修。

- 学校要占用教学楼进行考试。

  此时不允许教学楼中有正在上自习的教室，也不允许对教室进行维修。所以可以在教学楼门口放置一把 `X锁` （类似表级别的 `X锁` ）。此时：

  - 来上自习的学生们看到教学楼门口有 `X锁` ，则需要在教学楼门口等着，啥时候考试结束，把教学楼的 `X锁` 撤掉再进入教学楼上自习。
  - 修理工看到教学楼门口有 `X锁` ，则先在教学楼门口等着，啥时候考试结束，把教学楼的 `X锁` 撤掉再进入教学楼维修。

但是这里头有两个问题：

- 如果我们想对教学楼整体上 `S锁` ，首先需要确保教学楼中的没有正在维修的教室，如果有正在维修的教室，需要等到维修结束才可以对教学楼整体上 `S锁` 。
- 如果我们想对教学楼整体上 `X锁` ，首先需要确保教学楼中的没有上自习的教室以及正在维修的教室，如果有上自习的教室或者正在维修的教室，需要等到全部上自习的同学都上完自习离开，以及维修工维修完教室离开后才可以对教学楼整体上 `X锁` 。

我们在对教学楼整体上锁（ `表锁` ）时，怎么知道教学楼中有没有教室已经被上锁（ `行锁` ）了呢？依次检查每一间教室门口有没有上锁？那这效率也太慢了吧！遍历是不可能遍历的，这辈子也不可能遍历的，于是乎设计 `InnoDB` 的大叔们提出了一种称之为 `意向锁` （英文名： `Intention Locks` ）的东东：

- 意向共享锁，英文名： `Intention Shared Lock` ，简称 `IS锁` 。当事务准备在某条记录上加 `S锁` 时，需要先在表级别加一个 `IS锁` 。

- 意向独占锁，英文名： Intention Exclusive Lock ，简称 IX锁 。当事务准备在某条记录上加 X锁 时，需要先在表级别加一个 IX锁 。

视角回到教学楼和教室上来：

- 如果有学生到教室中上自习，那么他先在整栋教学楼门口放一把 IS锁 （表级锁），然后再到教室门口放一把 S锁 （行锁）。
- 如果有维修工到教室中维修，那么它先在整栋教学楼门口放一把 IX锁 （表级锁），然后再到教室门口放一把 X锁 （行锁）。

之后：

- 如果有领导要参观教学楼，也就是想在教学楼门口前放 S锁 （表锁）时，首先要看一下教学楼门口有没有 IX锁 ，如果有，意味着有教室在维修，需要等到维修结束把 IX锁 撤掉后才可以在整栋教学楼上加 S锁 。
- 如果有考试要占用教学楼，也就是想在教学楼门口前放 X锁 （表锁）时，首先要看一下教学楼门口有没有 IS锁 或 IX锁 ，如果有，意味着有教室在上自习或者维修，需要等到学生们上完自习以及维修结束把 IS锁 和 IX锁 撤掉后才可以在整栋教学楼上加 X锁 。

> 小贴士：
> 学生在教学楼门口加IS锁时，是不关心教学楼门口是否有IX锁的，维修工在教学楼门口加IX锁时，是不关心教学楼门口是否有IS锁或者其他IX锁的。IS和IX锁只是为了判断当前时间教学楼里有没有被占用的教室用的，也就是在对教学楼加S锁或者X锁时才会用到。

总结一下：IS、IX锁是表级锁，它们的提出仅仅为了在之后加表级别的S锁和X锁时可以快速判断表中的记录是否被上锁，以避免用遍历的方式来查看表中有没有上锁的记录，也就是说其实IS锁和IX锁是兼容的，IX锁和IX锁是兼容的。我们画个表来看一下表级别的各种锁的兼容性：

兼容性	X	IX	S	IS
X	不兼容	不兼容	不兼容	不兼容
IX	不兼容	兼容	不兼容	兼容
S	不兼容	不兼容	兼容	兼容
IS	不兼容	兼容	兼容	兼容

25.3 MySQL中的行锁和表锁

上边说的都算是些理论知识，其实 MySQL 支持多种存储引擎，不同存储引擎对锁的支持也是不一样的。当然，我们重点还是讨论 InnoDB 存储引擎中的锁，其他的存储引擎只是稍微提一下～

25.3.1 其他存储引擎中的锁

对于 MyISAM 、 MEMORY 、 MERGE 这些存储引擎来说，它们只支持表级锁，而且这些引擎并不支持事务，所以使用这些存储引擎的锁一般都是针对当前会话来说的。比方说在 Session 1 中对一个表执行 SELECT 操作，就相当于为这个表加了一个表级别的 S锁 ，如果在 SELECT 操作未完成时， Session 2 中对这个表执行 UPDATE 操作，相当于要获取表的 X锁 ，此操作会被阻塞，直到 Session 1 中的 SELECT 操作完成，释放掉表级别的 S锁 后， Session 2 中对这个表执行 UPDATE 操作才能继续获取 X锁 ，然后执行具体的更新语句。

> 小贴士：
> 因为使用MyISAM、MEMORY、MERGE这些存储引擎的表在同一时刻只允许一个会话对表进行写操作，所以这些存储引擎实际上最好用在只读，或者大部分都是读操作，或者单用户的情景下。
> 另外，在MyISAM存储引擎中有一个称之为Concurrent Inserts的特性，支持在对MyISAM表读取时同时插入记录，这样可以提升一些插入速度。关于更多Concurrent Inserts的细节，我们就不唠叨了，详情可以参考文档。

25.3.2 InnoDB存储引擎中的锁

InnoDB 存储引擎既支持表锁，也支持行锁。表锁实现简单，占用资源较少，不过粒度很粗，有时候你仅仅需要锁住几条记录，但使用表锁的话相当于为表中的所有记录都加锁，所以性能比较差。行锁粒度更细，可以实现更精准的并发控制。下边我们详细看一下。

25.3.2.1 InnoDB中的表级锁

- 表级别的 S锁 、 X锁

  在对某个表执行 SELECT 、 INSERT 、 DELETE 、 UPDATE 语句时， InnoDB 存储引擎是不会为这个表添加表级别的 S锁 或者 X锁 的。

  另外，在对某个表执行一些诸如 ALTER TABLE 、 DROP TABLE 这类的 DDL 语句时，其他事务对这个表并发执行诸如 SELECT 、 INSERT 、 DELETE 、 UPDATE 的语句会发生阻塞，同理，某个事务中对某个表执行 SELECT 、 INSERT 、 DELETE 、 UPDATE 语句时，在其他会话中对这个表执行 DDL 语句也会发生阻塞。这个过程其实是通过在 server层 使用一种称之为 元数据锁 （英文名： Metadata Locks ，简称 MDL ）东东来实现的，一般情况下也不会使用 InnoDB 存储引擎自己提供的表级别的 S锁 和 X锁 。

  > 小贴士：
  > 在事务简介的章节中我们说过，DDL语句执行时会隐式的提交当前会话中的事务，这主要是DDL语句的执行一般都会在若干个特殊事务中完成，在开启这些特殊事务前，需要将当前会话中的事务提交掉。另外，关于MDL锁并不是我们本章所要讨论的范围，大家可以参阅文档了解哈～

  其实这个 InnoDB 存储引擎提供的表级 S锁 或者 X锁 是相当鸡肋，只会在一些特殊情况下，比方说崩溃恢复过程中用到。不过我们还是可以手动获取一下的，比方说在系统变量 autocommit=0，innodb_table_locks = 1 时，手动获取 InnoDB 存储引擎提供的表 t 的 S锁 或者 X锁 可以这么写：
  - LOCK TABLES t READ ： InnoDB 存储引擎会对表 t 加表级别的 S锁 。
  - LOCK TABLES t WRITE ： InnoDB 存储引擎会对表 t 加表级别的 X锁 。

  不过请尽量避免在使用 InnoDB 存储引擎的表上使用 LOCK TABLES 这样的手动锁表语句，它们并不会提供什么额外的保护，只是会降低并发能力而已。 InnoDB 的厉害之处还是实现了更细粒度的行锁，关于表级别的 S锁 和 X锁 大家了解一下就罢了。

- 表级别的 IS锁 、 IX锁

  当我们在对使用 InnoDB 存储引擎的表的某些记录加 S锁 之前，那就需要先在表级别加一个 IS锁 ，当我们在对使用 InnoDB 存储引擎的表的某些记录加 X锁 之前，那就需要先在表级别加一个 IX锁 。 IS锁 和 IX锁 的使命只是为了后续在加表级别的 S锁 和 X锁 时判断表中是否有已经被加锁的记录，以避免用遍历的方式来查看表中有没有上锁的记录。更多关于 IS锁 和 IX锁 的解释我们上边都唠叨过了，就不赘述了。

- 表级别的 AUTO-INC锁

  在使用 MySQL 过程中，我们可以为表的某个列添加 AUTO_INCREMENT 属性，之后在插入记录时，可以不指定该列的值，系统会自动为它赋上递增的值，比方说我们有一个表：

  ```
  CREATE TABLE t (
      id INT NOT NULL AUTO_INCREMENT,
      c VARCHAR(100),
      PRIMARY KEY (id)
  ) Engine=InnoDB CHARSET=utf8;
  ```

  由于这个表的 id 字段声明了 AUTO_INCREMENT ，也就意味着在书写插入语句时不需要为其赋值，比方说这样：

  ```
  INSERT INTO t(c) VALUES('aa'), ('bb');
  ```

  上边的插入语句并没有为 id 列显式赋值，所以系统会自动为它赋上递增的值，效果就是这样：

```
mysql> SELECT * FROM t;
+----+------+
| id | c    |
+----+------+
|  1 | aa   |
|  2 | bb   |
+----+------+
2 rows in set (0.00 sec)
```

系统实现这种自动给 AUTO_INCREMENT 修饰的列递增赋值的原理主要是两个:

- 采用 AUTO-INC 锁，也就是在执行插入语句时就在表级别加一个 AUTO-INC 锁，然后为每条待插入记录的 AUTO_INCREMENT 修饰的列分配递增的值，在该语句执行结束后，再把 AUTO-INC 锁释放掉。这样一个事务在持有 AUTO-INC 锁的过程中，其他事务的插入语句都要被阻塞，可以保证一个语句中分配的递增值是连续的。

  如果我们的插入语句在执行前不可以确定具体要插入多少条记录（无法预计即将插入记录的数量），比方说使用 INSERT ... SELECT 、 REPLACE ... SELECT 或者 LOAD DATA 这种插入语句，一般是使用 AUTO-INC 锁为 AUTO_INCREMENT 修饰的列生成对应的值。

  > 小贴士：
  > 需要注意一下的是，这个AUTO-INC锁的作用范围只是单个插入语句，插入语句执行完成后，这个锁就被释放了，跟我们之前介绍的锁在事务结束时释放是不一样的。

- 采用一个轻量级的锁，在为插入语句生成 AUTO_INCREMENT 修饰的列的值时获取一下这个轻量级锁，然后生成本次插入语句需要用到的 AUTO_INCREMENT 列的值之后，就把该轻量级锁释放掉，并不需要等到整个插入语句执行完才释放锁。

  如果我们的插入语句在执行前就可以确定具体要插入多少条记录，比方说我们上边举的关于表 t 的例子中，在语句执行前就可以确定要插入2条记录，那么一般采用轻量级锁的方式对 AUTO_INCREMENT 修饰的列进行赋值。这种方式可以避免锁定表，可以提升插入性能。

  > 小贴士：
  > 设计InnoDB的大叔提供了一个称之为innodb_autoinc_lock_mode的系统变量来控制到底使用上述两种方式中的哪种来为AUTO_INCREMENT修饰的列进行赋值，当innodb_autoinc_lock_mode值为0时，一律采用AUTO-INC锁；当innodb_autoinc_lock_mode值为2时，一律采用轻量级锁；当innodb_autoinc_lock_mode值为1时，两种方式混着来（也就是在插入记录数量确定时采用轻量级锁，不确定时使用AUTO-INC锁）。不过当innodb_autoinc_lock_mode值为2时，可能会造成不同事务中的插入语句为AUTO_INCREMENT修饰的列生成的值是交叉的，在有主从复制的场景中是不安全的。

25.3.2.2 InnoDB中的行级锁

很遗憾的通知大家一个不好的消息，上边讲的都是铺垫，本章真正的重点才刚刚开始[手动偷笑]。

行锁 ，也称为 记录锁 ，顾名思义就是在记录上加的锁。不过设计 InnoDB 的大叔很有才，一个 行锁 玩出了各种花样，也就是把 行锁 分成了各种类型。换句话说即使对同一条记录加 行锁 ，如果类型不同，起到的功效也是不同的。为了故事的顺利发展，我们还是先将之前唠叨 MVCC 时用到的表抄一遍:

```
CREATE TABLE hero (
    number INT,
    name VARCHAR(100),
    country varchar(100),
    PRIMARY KEY (number),
    KEY idx_name (name)
) Engine=InnoDB CHARSET=utf8;
```

我们主要是想用这个表存储三国时的英雄，然后向这个表里插入几条记录：

```
INSERT INTO hero VALUES
    (1, 'l刘备', '蜀'),
    (3, 'z诸葛亮', '蜀'),
    (8, 'c曹操', '魏'),
    (15, 'x荀彧', '魏'),
    (20, 's孙权', '吴');
```

现在表里的数据就是这样的：

```
mysql> SELECT * FROM hero;
+--------+------------+---------+
| number | name       | country |
+--------+------------+---------+
|      1 | l刘备      | 蜀      |
|      3 | z诸葛亮    | 蜀      |
|      8 | c曹操      | 魏      |
|     15 | x荀彧      | 魏      |
|     20 | s孙权      | 吴      |
+--------+------------+---------+
5 rows in set (0.01 sec)
```

> 小贴士：
> 不是说好的存储三国时的英雄么，你在搞什么，为啥要在'刘备'、'曹操'、'孙权'前边加上'l'、'c'、's'这几个字母呀？这个主要是因为我们采用utf8字符集，该字符集并没有对应的按照汉语拼音进行排序的比较规则，也就是说'刘备'、'曹操'、'孙权'这几个字符串的排序并不是按照它们汉语拼音进行排序的，我怕大家懵逼，所以在汉字前边加上了汉字对应的拼音的第一个字母，这样在排序时就是按照汉语拼音进行排序，大家也不懵逼了。
> 另外，我们故意把各条记录number列的值搞得很分散，后边会用到，稍安勿躁哈～

我们把 hero 表中的聚簇索引的示意图画一下：

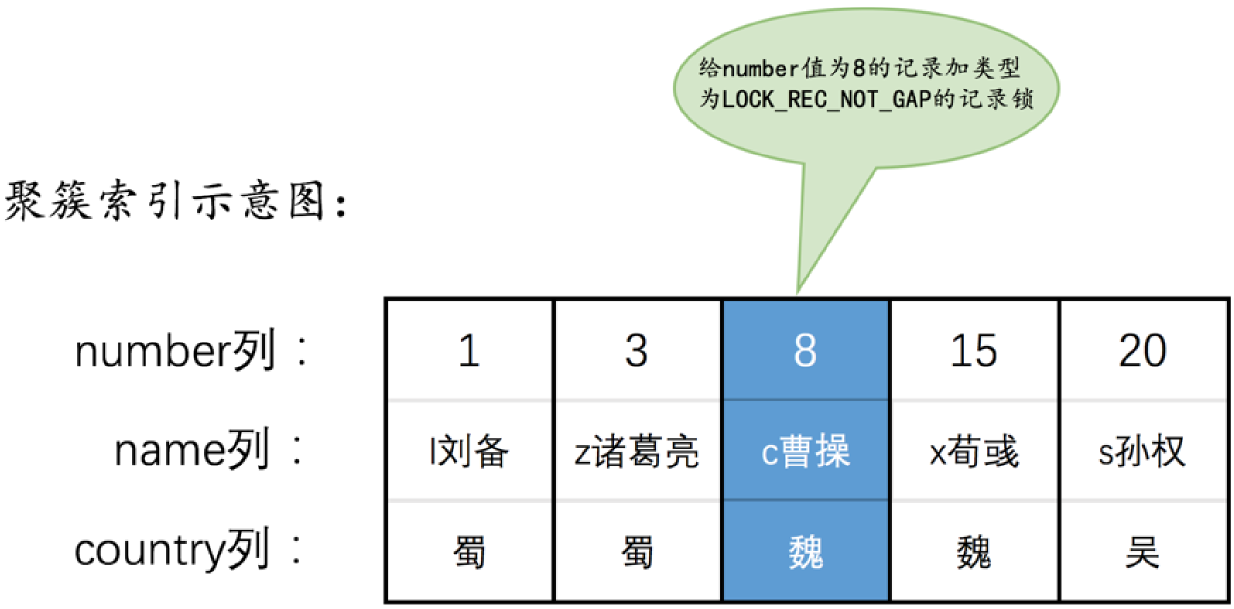


当然，我们把 B+树 的索引结构做了一个超级简化，只把索引中的记录给拿了出来，我们这里只是想强调聚簇索引中的记录是按照主键大小排序的，并且省略掉了聚簇索引中的隐藏列，大家心里明白就好（不理解索引结构的同学可以去前边的文章中查看）。

现在准备工作做完了，下边我们来看看都有哪些常用的 行锁类型 。

- Record Locks ：

  我们前边提到的记录锁就是这种类型，也就是仅仅把一条记录锁上，我决定给这种类型的锁起一个比较不正经的名字： 正经记录锁 （请允许我皮一下，我实在不知道该叫个啥名好）。官方的类型名称为： LOCK_REC_NOT_GAP 。比方说我们把 number 值为 8 的那条记录加一个 正经记录锁 的示意图如下：

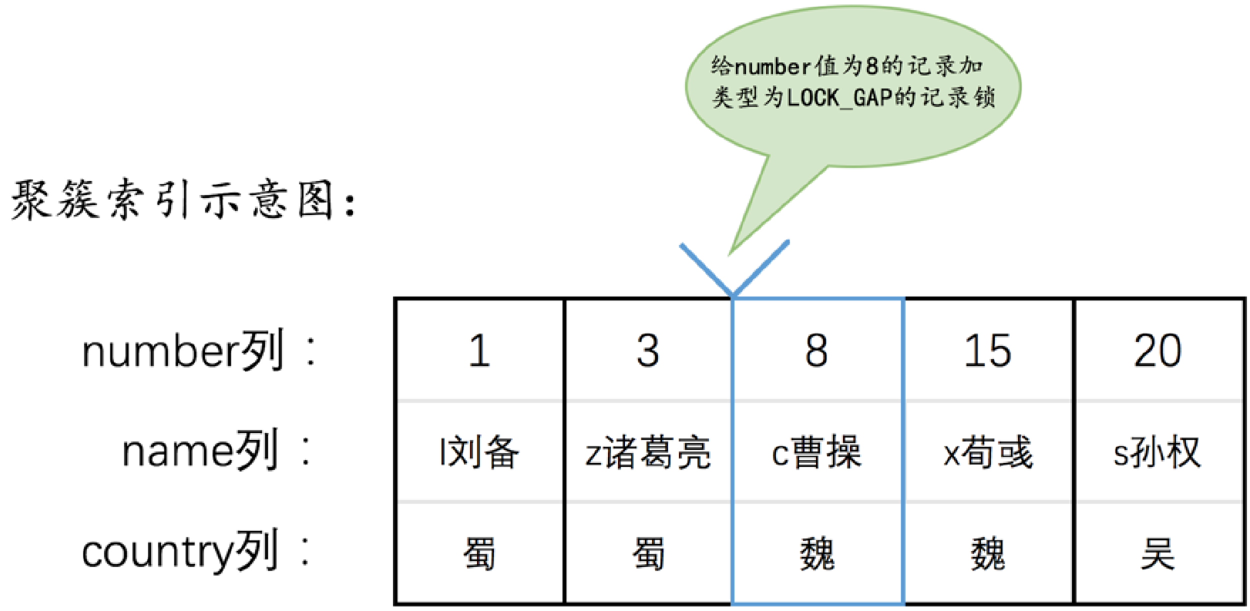


  正经记录锁 是有 S锁 和 X锁 之分的，让我们分别称之为 S型正经记录锁 和 X型正经记录锁 吧（听起来有点怪怪的），当一个事务获取了一条记录的 S型正经记录锁 后，其他事务也可以继续获取该记录的 S型正经记录锁 ，但不可以继续获取 X型正经记录锁 ；当一个事务获取了一条记录的 X型正经记录锁 后，其他事务既不可以继续获取该记录的 S型正经记录锁 ，也不可以继续获取 X型正经记录锁 ；

- Gap Locks ：

  我们说 MySQL 在 REPEATABLE READ 隔离级别下是可以解决幻读问题的，解决方案有两种，可以使用 MVCC 方案解决，也可以采用 加锁 方案解决。但是在使用 加锁 方案解决时有个大问题，就是事务在第一次执行读取操作时，那些幻影记录尚不存在，我们无法给这些幻影记录加上 正经记录锁 。不过这难不倒设计 InnoDB 的

大叔，他们提出了一种称之为 Gap Locks 的锁，官方的类型名称为： LOCK_GAP ，我们也可以简称为 gap锁 。比方说我们把 number 值为 8 的那条记录加一个 gap锁 的示意图如下：



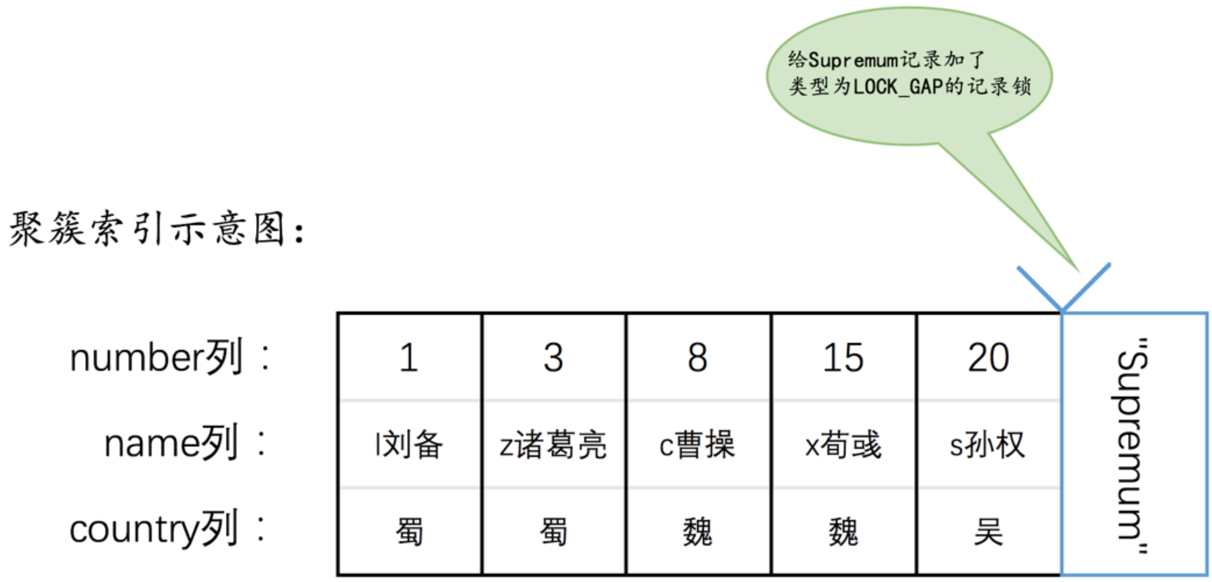
如图中为 number 值为 8 的记录加了 gap锁 ，意味着不允许别的事务在 number 值为 8 的记录前边的 间隙 插入新记录，其实就是 number 列的值 (3, 8) 这个区间的新记录是不允许立即插入的。比方说有另外一个事务再想插入一条 number 值为 4 的新记录，它定位到该条新记录的下一条记录的 number 值为8，而这条记录上又有一个 gap锁 ，所以就会阻塞插入操作，直到拥有这个 gap锁 的事务提交了之后， number 列的值在区间 (3, 8) 中的新记录才可以被插入。

这个 gap锁 的提出仅仅是为了防止插入幻影记录而提出的，虽然有 共享gap锁 和 独占gap锁 这样的说法，但是它们起到的作用都是相同的。而且如果你对一条记录加了 gap锁 （不论是 共享gap锁 还是 独占gap锁 ），并不会限制其他事务对这条记录加 正经记录锁 或者继续加 gap锁 ，再强调一遍， gap锁 的作用仅仅是为了防止插入幻影记录的而已。

不知道大家发现了一个问题没，给一条记录加了 gap锁 只是不允许其他事务往这条记录前边的间隙插入新记录，那对于最后一条记录之后的间隙，也就是 hero 表中 number 值为 20 的记录之后的间隙该咋办呢？也就是说给哪条记录加 gap锁 才能阻止其他事务插入 number 值在 (20, +∞) 这个区间的新记录呢？这时候应该想起我们在前边唠叨 数据页 时介绍的两条伪记录了：

- Infimum 记录，表示该页面中最小的记录。
- Supremum 记录，表示该页面中最大的记录。

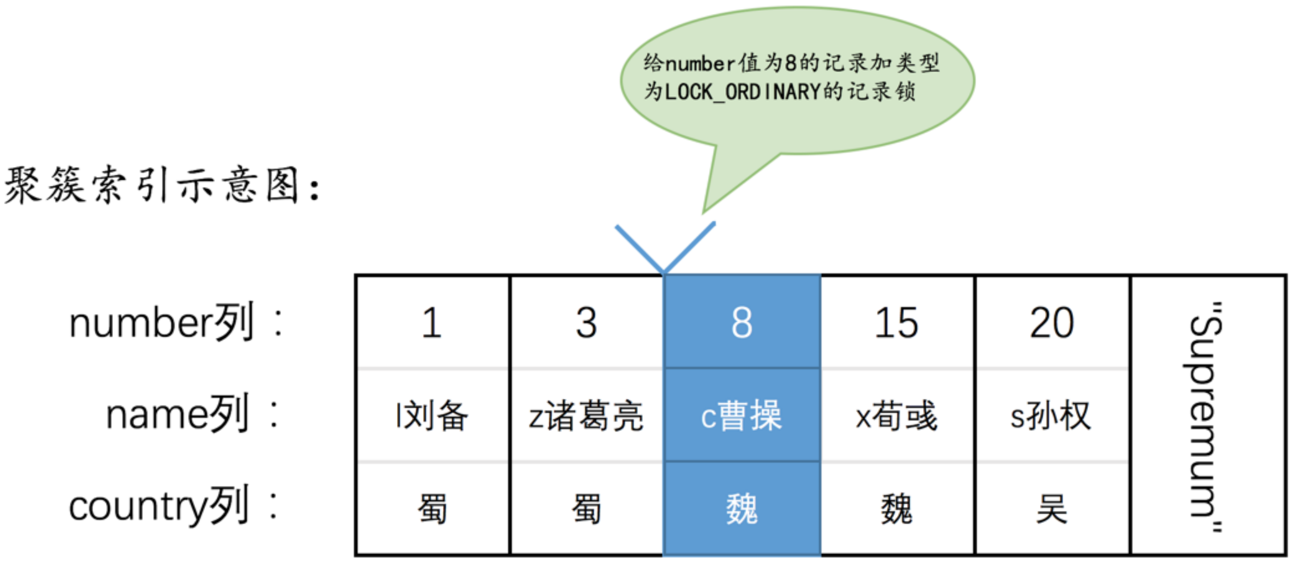
为了实现阻止其他事务插入 number 值在 (20, +∞) 这个区间的新记录，我们可以给索引中的最后一条记录，也就是 number 值为 20 的那条记录所在页面的 Supremum 记录加上一个 gap锁 ，画个图就是这样：



这样就可以阻止其他事务插入 number 值在（20, +∞）这个区间的新记录。为了大家理解方便，之后的索引示意图中都会把这个 Supremum 记录画出来。

- Next-Key Locks ：

有时候我们既想锁住某条记录，又想阻止其他事务在该记录前边的 间隙 插入新记录，所以设计 InnoDB 的大叔们就提出了一种称之为 Next-Key Locks 的锁，官方的类型名称为： LOCK_ORDINARY ，我们也可以简称为 next-key锁 。比方说我们把 number 值为 8 的那条记录加一个 next-key锁 的示意图如下：

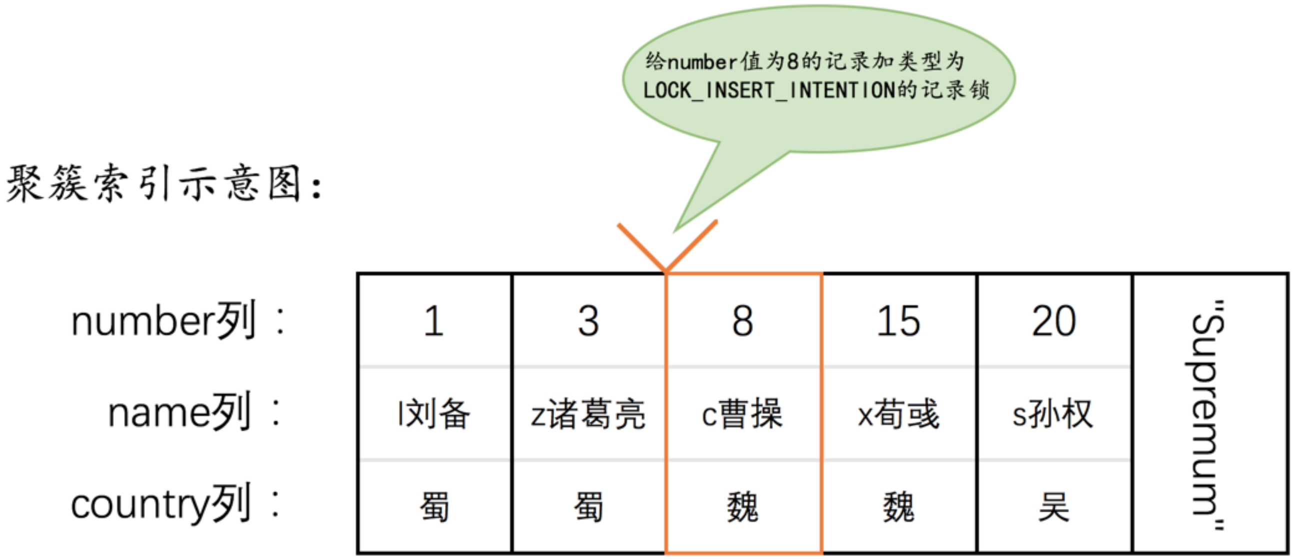


next-key锁 的本质就是一个 正经记录锁 和一个 gap锁 的合体，它既能保护该条记录，又能阻止别的事务将新记录插入被保护记录前边的 间隙 。

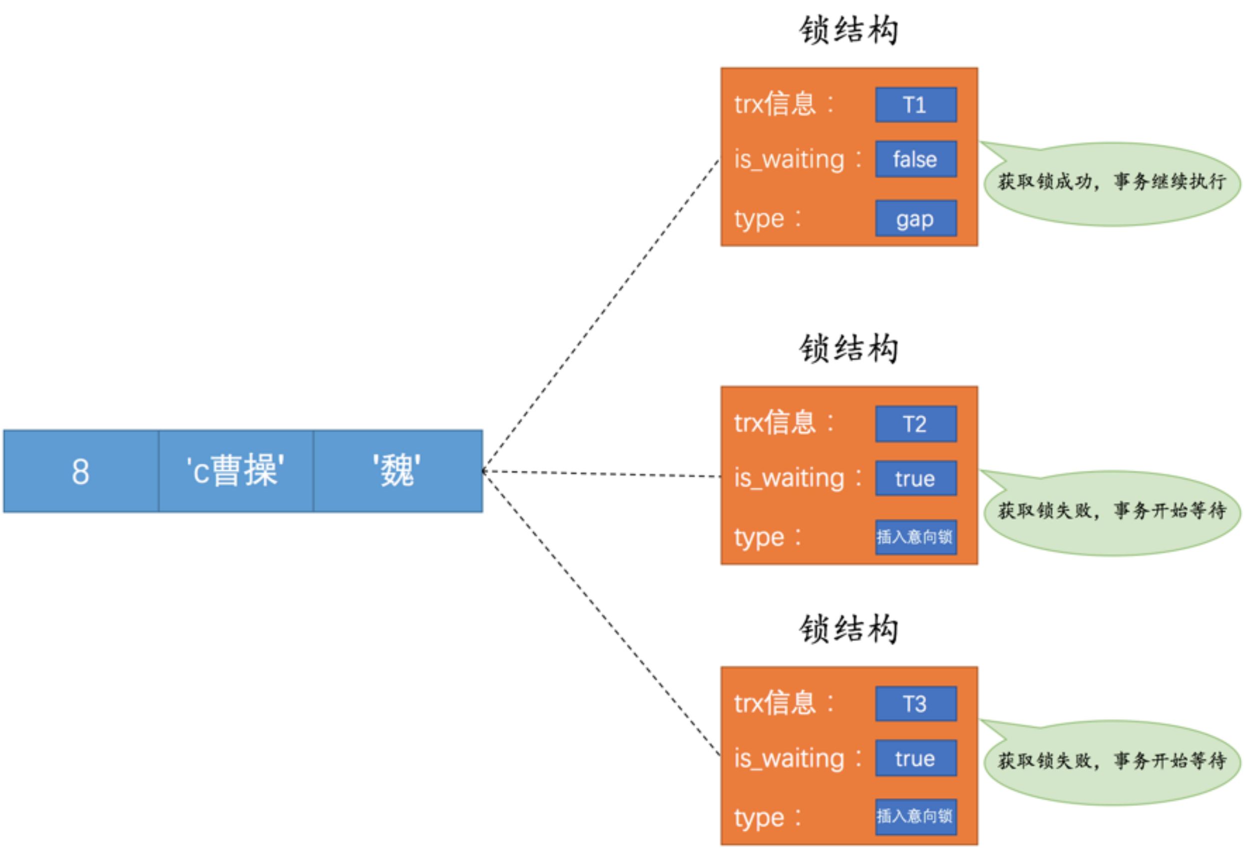
- Insert Intention Locks ：

我们说一个事务在插入一条记录时需要判断一下插入位置是不是被别的事务加了所谓的 gap锁 （ next-key锁 也包含 gap锁 ，后边就不强调了），如果有的话，插入操作需要等待，直到拥有 gap锁 的那个事务提交。但是设计 InnoDB 的大叔规定事务在等待的时候也需要在内存中生成一个 锁结构 ，表明有事务想在某个 间隙 中插入新记录，但是现在在等待。设计 InnoDB 的大叔就把这种类型的锁命名为 Insert Intention Locks ，官方的类型名称为： LOCK_INSERT_INTENTION ，我们也可以称为 插入意向锁 。

比方说我们把 number 值为 8 的那条记录加一个 插入意向锁 的示意图如下：



为了让大家彻底理解这个 插入意向锁 的功能，我们还是举个例子然后画个图表示一下。比方说现在 T1 为 number 值为 8 的记录加了一个 gap锁 ，然后 T2 和 T3 分别想向 hero 表中插入 number 值分别为 4 、 5 的两条记录，所以现在为 number 值为 8 的记录加的锁的示意图就如下所示：



> 小贴士：
> 我们在锁结构中又新添了一个type属性，表明该锁的类型。稍后会全面介绍InnoDB存储引擎中的一个锁结构到底长什么样。

从图中可以看到，由于 T1 持有 gap锁 ，所以 T2 和 T3 需要生成一个 插入意向锁 的 锁结构 并且处于等待状态。当 T1 提交后会把它获取到的锁都释放掉，这样 T2 和 T3 就能获取到对应的 插入意向锁 了（本质上就是把插入意向锁对应锁结构的 is_waiting 属性改为 false ）， T2 和 T3 之间也并不会相互阻塞，它们可以同时获取到 number 值为8的 插入意向锁 ，然后执行插入操作。事实上插入意向锁并不会阻止别的事务继续获取该记录上任何类型的锁（ 插入意向锁 就是这么鸡肋）。

- 隐式锁

隐式锁

我们前边说一个事务在执行 INSERT 操作时，如果即将插入的 间隙 已经被其他事务加了 gap锁 ，那么本次 INSERT 操作会阻塞，并且当前事务会在该间隙上加一个 插入意向锁 ，否则一般情况下 INSERT 操作是不加锁的。那如果一个事务首先插入了一条记录（此时并没有与该记录关联的锁结构），然后另一个事务：

- 立即使用 SELECT ... LOCK IN SHARE MODE 语句读取这条事务，也就是在要获取这条记录的 S锁 ，或者使用 SELECT ... FOR UPDATE 语句读取这条事务或者直接修改这条记录，也就是要获取这条记录的 X锁 ，该咋办？

  如果允许这种情况的发生，那么可能产生 脏读 问题。
- 立即修改这条记录，也就是要获取这条记录的 X锁 ，该咋办？

  如果允许这种情况的发生，那么可能产生 脏写 问题。

  这时候我们前边唠叨了很多遍的 事务id 又要起作用了。我们把聚簇索引和二级索引中的记录分开看一下：
- 情景一：对于聚簇索引记录来说，有一个 trx_id 隐藏列，该隐藏列记录着最后改动该记录的 事务id 。那么如果在当前事务中新插入一条聚簇索引记录后，该记录的 trx_id 隐藏列代表的的就是当前事务的 事务id ，如果其他事务此时想对该记录添加 S锁 或者 X锁 时，首先会看一下该记录的 trx_id 隐藏列代表的事务是否是当前的活跃事务，如果是的话，那么就帮助当前事务创建一个 X锁 （也就是为当前事务创建一个锁结构， is_waiting 属性是 false ），然后自己进入等待状态（也就是为自己也创建一个锁结构， is_waiting 属性是 true ）。
- 情景二：对于二级索引记录来说，本身并没有 trx_id 隐藏列，但是在二级索引页面的 Page Header 部分有一个 PAGE_MAX_TRX_ID 属性，该属性代表对该页面做改动的最大的 事务id ，如果 PAGE_MAX_TRX_ID 属性值小于当前最小的活跃 事务id ，那么说明对该页面做修改的事务都已经提交了，否则就需要在页面中定位到对应的二级索引记录，然后回表找到它对应的聚簇索引记录，然后再重复 情景一 的做法。

  通过上边的叙述我们知道，一个事务对新插入的记录可以不显式的加锁（生成一个锁结构），但是由于 事务id 这个牛逼的东东的存在，相当于加了一个 隐式锁 。别的事务在对这条记录加 S锁 或者 X锁 时，由于 隐式锁 的存在，会先帮助当前事务生成一个锁结构，然后自己再生成一个锁结构后进入等待状态。

  > 小贴士：
  > 除了插入意向锁，在一些特殊情况下INSERT还会获取一些锁，我们稍后唠叨哈。

25.3.3 InnoDB锁的内存结构

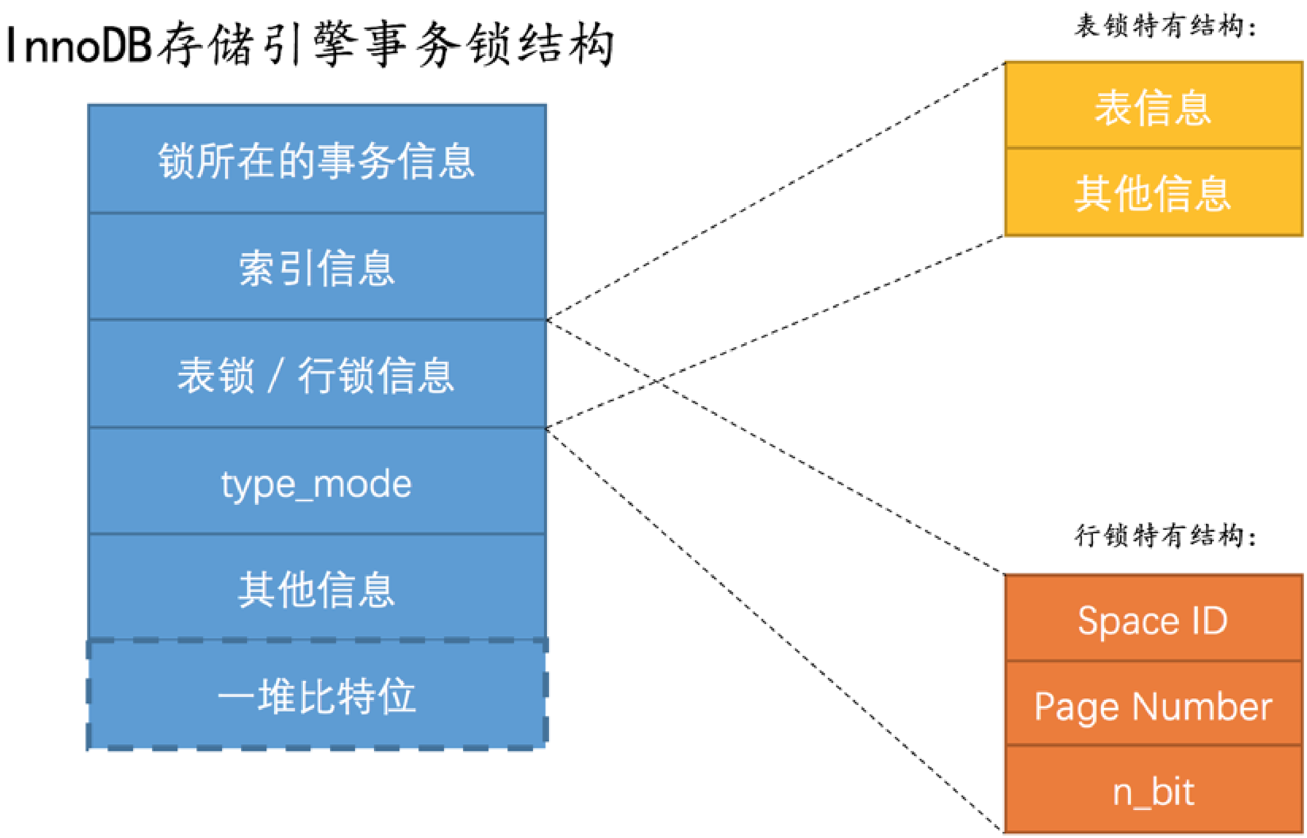
我们前边说对一条记录加锁的本质就是在内存中创建一个 锁结构 与之关联，那么是不是一个事务对多条记录加锁，就要创建多个 锁结构 呢？比方说事务 T1 要执行下边这个语句：

```
# 事务T1
SELECT * FROM hero LOCK IN SHARE MODE;
```

很显然这条语句需要为 hero表 中的所有记录进行加锁，那是不是需要为每条记录都生成一个 锁结构 呢？其实理论上创建多个 锁结构 没问题，反而更容易理解，但是谁知道你在一个事务里想对多少记录加锁呢，如果一个事务要获取10000条记录的锁，要生成10000个这样的结构也太亏了吧！所以设计 InnoDB 的大叔本着勤俭节约的传统美德，决定在对不同记录加锁时，如果符合下边这些条件：

- 在同一个事务中进行加锁操作
- 被加锁的记录在同一个页面中
- 加锁的类型是一样的
- 等待状态是一样的

那么这些记录的锁就可以被放到一个 锁结构 中。当然，这么空口白牙的说有点儿抽象，我们还是画个图来看看 InnoDB 存储引擎中的 锁结构 具体长啥样吧：



我们看看这个结构里边的各种信息都是干嘛的：

- `锁所在的事务信息` ：

  不论是 `表锁` 还是 `行锁` ，都是在事务执行过程中生成的，哪个事务生成了这个 `锁结构` ，这里就记载着这个事务的信息。

  > 小贴士：
  > 实际上这个所谓的`锁所在的事务信息`在内存结构中只是一个指针而已，所以不会占用多大内存空间，通过指针可以找到内存中关于该事务的更多信息，比方说事务id是什么。下边介绍的所谓的`索引信息`其实也是一个指针。

- `索引信息` ：

  对于 `行锁` 来说，需要记录一下加锁的记录是属于哪个索引的。
- `表锁 / 行锁信息` ：

  `表锁结构` 和 `行锁结构` 在这个位置的内容是不同的：
  - 表锁：

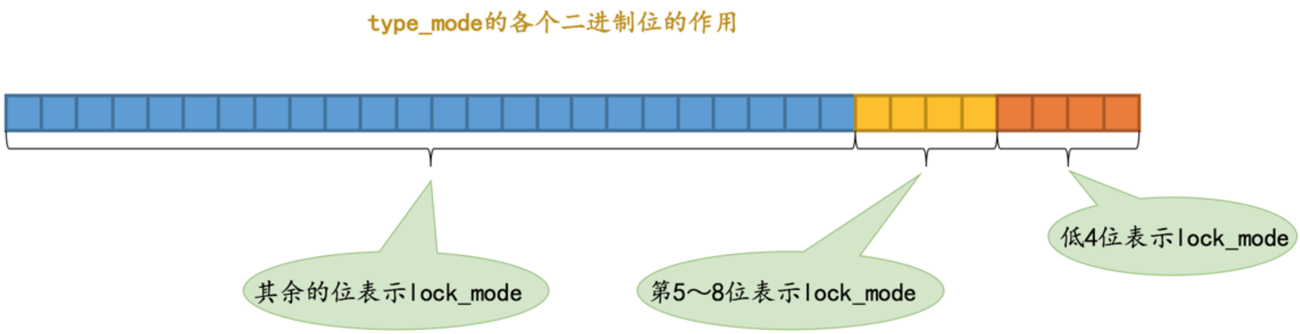
    记载着这是对哪个表加的锁，还有其他的一些信息。
  - 行锁：

    记载了三个重要的信息：
    - `Space ID` ：记录所在表空间。
    - `Page Number` ：记录所在页号。
    - `n_bits` ：对于行锁来说，一条记录就对应着一个比特位，一个页面中包含很多记录，用不同的比特位来区分到底是哪一条记录加了锁。为此在行锁结构的末尾放置了一堆比特位，这个 `n_bits` 属性代表使用了多少比特位。

> 小贴士：
> 并不是该页面中有多少记录，n_bits属性的值就是多少。为了让之后在页面中插入了新记录后也不至于重新分配锁结构，所以n_bits的值一般都比页面中记录条数多一些。

- type_mode ：

这是一个32位的数，被分成了 lock_mode 、 lock_type 和 rec_lock_type 三个部分，如图所示：



  - 锁的模式（ lock_mode ），占用低4位，可选的值如下：
    - LOCK_IS （十进制的 0 ）：表示共享意向锁，也就是 IS锁 。
    - LOCK_IX （十进制的 1 ）：表示独占意向锁，也就是 IX锁 。
    - LOCK_S （十进制的 2 ）：表示共享锁，也就是 S锁 。
    - LOCK_X （十进制的 3 ）：表示独占锁，也就是 X锁 。
    - LOCK_AUTO_INC （十进制的 4 ）：表示 AUTO-INC锁 。

> 小贴士：
> 在InnoDB存储引擎中，LOCK_IS，LOCK_IX，LOCK_AUTO_INC都算是表级锁的模式，LOCK_S和LOCK_X既可以算是表级锁的模式，也可以是行级锁的模式。

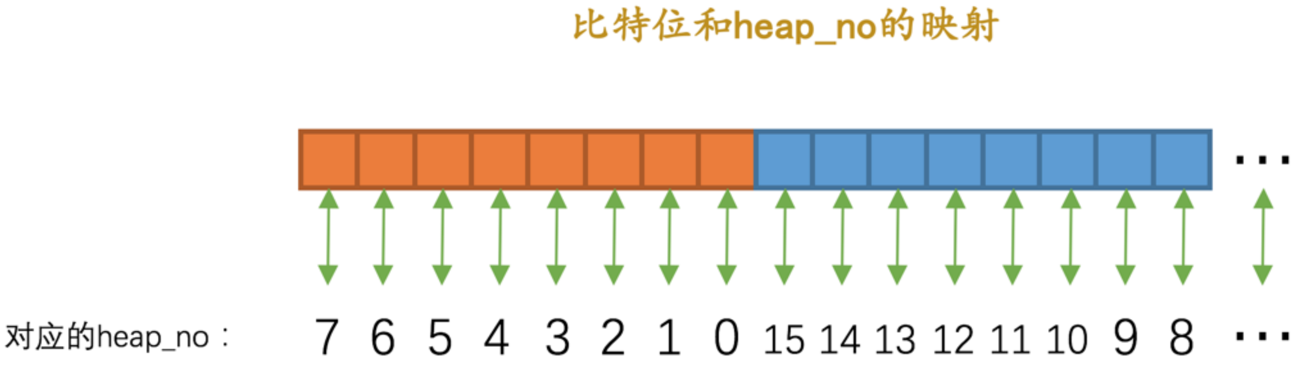
  - 锁的类型（ lock_type ），占用第5～8位，不过现阶段只有第5位和第6位被使用：
    - LOCK_TABLE （十进制的 16 ），也就是当第5个比特位置为1时，表示表级锁。
    - LOCK_REC （十进制的 32 ），也就是当第6个比特位置为1时，表示行级锁。
  - 行锁的具体类型（ rec_lock_type ），使用其余的位来表示。只有在 lock_type 的值为 LOCK_REC 时，也就是只有在该锁为行级锁时，才会被细分为更多的类型：
    - LOCK_ORDINARY （十进制的 0 ）：表示 next-key锁 。
    - LOCK_GAP （十进制的 512 ）：也就是当第10个比特位置为1时，表示 gap锁 。
    - LOCK_REC_NOT_GAP （十进制的 1024 ）：也就是当第11个比特位置为1时，表示 正经记录锁 。
    - LOCK_INSERT_INTENTION （十进制的 2048 ）：也就是当第12个比特位置为1时，表示插入意向锁。
    - 其他的类型：还有一些不常用的类型我们就不多说了。

    怎么还没看见 is_waiting 属性呢？这主要还是设计 InnoDB 的大叔太抠门了，一个比特位也不想浪费，所以他们把 is_waiting 属性也放到了 type_mode 这个32位的数字中：

    - LOCK_WAIT （十进制的 256 ） ：也就是当第9个比特位置为 1 时，表示 is_waiting 为 true ，也就是当前事务尚未获取到锁，处在等待状态；当这个比特位为 0 时，表示 is_waiting 为 false ，也就是当前事务获取锁成功。
- 其他信息 ：

为了更好的管理系统运行过程中生成的各种锁结构而设计了各种哈希表和链表，为了简化讨论，我们忽略这部分信息哈～
- 一堆比特位 ：

如果是 行锁结构 的话，在该结构末尾还放置了一堆比特位，比特位的数量是由上边提到的 n_bits 属性表示的。我们前边唠叨InnoDB记录结构的时候说过，页面中的每条记录在 记录头信息 中都包含一个 heap_no 属性，伪记录 Infimum 的 heap_no 值为 0 ， Supremum 的 heap_no 值为 1 ，之后每插入一条记录， heap_no 值就增1。 锁结构 最后的一堆比特位就对应着一个页面中的记录，一个比特位映射一个 heap_no ，不过为了编码方便，映射方式有点怪：



> 小贴士：
> 这么怪的映射方式纯粹是为了敲代码方便，大家不要大惊小怪，只需要知道一个比特位映射到页内的一条记录就好了。

可能上边的描述大家觉得还是有些抽象，我们还是举个例子说明一下。比方说现在有两个事务 T1 和 T2 想对 hero 表中的记录进行加锁， hero 表中记录比较少，假设这些记录都存储在所在的表空间号为 67 ，页号为 3 的页面上，那么如果：

- T1 想对 number 值为 15 的这条记录加 S型正常记录锁 ，在对记录加行锁之前，需要先加表级别的 IS 锁，也就是会生成一个表级锁的内存结构，不过我们这里不关心表级锁，所以就忽略掉了哈～ 接下来分析一下生成行锁结构的过程：
  - 事务 T1 要进行加锁，所以锁结构的 锁所在事务信息 指的就是 T1 。
  - 直接对聚簇索引进行加锁，所以索引信息指的其实就是 PRIMARY 索引。
  - 由于是行锁，所以接下来需要记录的是三个重要信息：
    - Space ID ：表空间号为 67 。
    - Page Number ：页号为 3 。
    - n_bits ：我们的 hero 表中现在只插入了5条用户记录，但是在初始分配比特位时会多分配一些，这主要是为了在之后新增记录时不用频繁分配比特位。其实计算 n_bits 有一个公式：

      ```
      n_bits = (1 + ((n_recs + LOCK_PAGE_BITMAP_MARGIN) / 8)) * 8
      ```

      其中 n_recs 指的是当前页面中一共有多少条记录（算上伪记录和在垃圾链表中的记录），比方说现在 hero 表一共有7条记录（5条用户记录和2条伪记录），所以 n_recs 的值就是 7 ， LOCK_PAGE_BITMAP_MARGIN 是一个固定的值 64 ，所以本次加锁的 n_bits 值就是：

      ```
      n_bits = (1 + ((7 + 64) / 8)) * 8 = 72
      ```

    - type_mode 是由三部分组成的：
      - lock_mode ，这是对记录加 S锁 ，它的值为 LOCK_S 。
      - lock_type ，这是对记录进行加锁，也就是行锁，所以它的值为 LOCK_REC 。
      - rec_lock_type ，这是对记录加 正经记录锁 ，也就是类型为 LOCK_REC_NOT_GAP 的锁。另外，由于当前没有其他事务对该记录加锁，所以应当获取到锁，也就是 LOCK_WAIT 代表的二进制位应该是0。

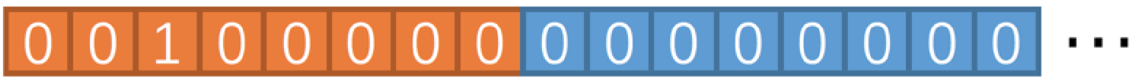
      综上所属，此次加锁的 type_mode 的值应该是：

```
type_mode = LOCK_S | LOCK_REC | LOCK_REC_NOT_GAP
也就是：
type_mode = 2 | 32 | 1024 = 1058
```

- 其他信息

  略～
- 一堆比特位

  因为 number 值为 15 的记录 heap_no 值为 5 ，根据上边列举的比特位和 heap_no 的映射图来看，应该是第一个字节从低位往高位数第6个比特位被置为1，就像这样：



综上所述，事务 T1 为 number 值为5的记录加锁生成的锁结构就如下图所示：



- T2 想对 number 值为 3 、 8 、 15 的这三条记录加 X型的next-key锁 ，在对记录加行锁之前，需要先加表级别的 IX 锁，也就是会生成一个表级锁的内存结构，不过我们这里不关心表级锁，所以就忽略掉了哈～

现在 T2 要为3条记录加锁， number 为 3 、 8 的两条记录由于没有其他事务加锁，所以可以成功获取这条记录的 X型next-key锁 ，也就是生成的锁结构的 is_waiting 属性为 false ；但是 number 为 15 的记录已经被 T1 加了 S型正经记录锁 ， T2 是不能获取到该记录的 X型next-key锁 的，也就是生成的锁结构的 is_waiting 属性为 true 。因为等待状态不相同，所以这时候会生成两个 锁结构 。这两个锁结构中相同的属性如下:

- 事务 T2 要进行加锁，所以锁结构的 锁所在事务信息 指的就是 T2 。
- 直接对聚簇索引进行加锁，所以索引信息指的其实就是 PRIMARY 索引。
- 由于是行锁，所以接下来需要记录是三个重要信息:
    - Space ID ：表空间号为 67 。
    - Page Number ：页号为 3 。
    - n_bits ：此属性生成策略同 T1 中一样，该属性的值为 72 。
    - type_mode 是由三部分组成的:
        - lock_mode ，这是对记录加 X锁 ，它的值为 LOCK_X 。
        - lock_type ，这是对记录进行加锁，也就是行锁，所以它的值为 LOCK_REC 。
        - rec_lock_type ，这是对记录加 next-key锁 ，也就是类型为 LOCK_ORDINARY 的锁。

- 其他信息

    略～

不同的属性如下：

- 为`number`为`3`、`8`的记录生成的`锁结构`：

    - `type_mode`值。

        由于可以获取到锁，所以`is_waiting`属性为`false`，也就是`LOCK_WAIT`代表的二进制位被置0。所以：
        ```
        type_mode = LOCK_X | LOCK_REC |LOCK_ORDINARY
        也就是
        type_mode = 3 | 32 | 0 = 35
        ```

    - `一堆比特位`

        因为`number`值为`3`、`8`的记录`heap_no`值分别为`3`、`4`，根据上边列举的比特位和`heap_no`的映射图来看，应该是第一个字节从低位往高位数第4、5个比特位被置为1，就像这样：

    ![image_1d9krhp4f1gd7hb4nhv1j182cb2q.png-21.2kB][17]

    综上所述，事务`T2`为`number`值为`3`、`8`两条记录加锁生成的锁结构就如下图所示：

    ![image_1d9krl3im1qr2tb4k1810bs18ak37.png-40.4kB][18]

- 为`number`为`15`的记录生成的`锁结构`：

    - `type_mode`值。

        由于可以获取到锁，所以`is_waiting`属性为`true`，也就是`LOCK_WAIT`代表的二进制位被置1。所以：
        ```
        type_mode = LOCK_X | LOCK_REC |LOCK_ORDINARY | LOCK_WAIT
        也就是
        type_mode = 3 | 32 | 0 | 256 = 291
        ```

    - `一堆比特位`

        因为`number`值为`15`的记录`heap_no`值为`5`，根据上边列举的比特位和`heap_no`的映射图来看，应该是第一个字节从低位往高位数第6个比特位被置为1，就像这样：

    ![image_1d9krpp171m7r2prc8cnhu1hkf3k.png-20.5kB][19]

    综上所述，事务`T2`为`number`值为`15`的记录加锁生成的锁结构就如下图所示：

    ![image_1d9krv36o145ub7vdr4cap16bq4h.png-43.4kB][20]

综上所述，事务 T1 先获取 number 值为 15 的 S型正经记录锁 ，然后事务 T2 获取 number 值为 3 、 8 、 15 的 X型正经记录锁 共需要生成3个锁结构。噗～ 关于锁结构我本来就想写一点点的，没想到一些起来就停不下了，大家乐呵乐呵看哈～

> 小贴士：
> 上边事务T2在对number值分别为3、8、15这三条记录加锁的情景中，是按照先对number值为3的记录加锁、再对number值为8的记录加锁，最后对number值为15的记录加锁的顺序进行的，如果我们一开始就对number值为15的记录加锁，那么该事务在为number值为15的记录生成一个锁结构后，直接就进入等待状态，就不为number值为3、8的两条记录生成锁结构了。在事务T1提交后会把在number值为15的记录上获取的锁释放掉，然后事务T2就可以获取该记录上的锁，这时再对number值为3、8的两条记录加锁时，就可以复用之前为number值为15的记录加锁时生成的锁结构了。

25.4 更多内容

欢迎各位关注我的微信公众号 「我们都是小青蛙」，那里有更多技术干货与特色扯犊子文章（后续会在公众号中发布各种不同的语句具体的加锁情况分析，敬请期待）。

26 第26章 写作本书时用到的一些重要的参考资料

26.1 感谢

我不生产知识，只是知识的搬运工。写作本小册的时间主要用在了两个方面：

- 搞清楚事情的本质是什么。

  这个过程就是研究源码、书籍和资料。
- 如何把我已经知道的知识表达出来。

  这个过程就是我不停的在地上走过来走过去，梳理知识结构，斟酌用词用句，不停的将已经写好的文章推倒重来，只是想给大家一个不错的用户体验。

这两个方面用的时间基本上是一半一半吧，在搞清楚事情的本质是什么阶段，除了直接阅读 MySQL 的源码之外，查看参考资料也是一种比较偷懒的学习方式。本书只是 MySQL 进阶的一个入门，想了解更多关于 MySQL 的知识，大家可以从下边这些资料里找点灵感。

26.1.1 一些链接

- MySQL官方文档：https://dev.mysql.com/doc/refman/5.7/en/ (https://dev.mysql.com/doc/refman/5.7/en/)

  MySQL 官方文档是写作本书时参考最多的一个资料。说实话，文档写的非常通俗易懂，唯一的缺点就是太长了，导致大家看的时候无从下手。
- MySQL Internals Manual：https://dev.mysql.com/doc/internals/en/ (https://dev.mysql.com/doc/internals/en/)

  介绍MySQL如何实现各种功能的文档，写的比较好，但是太少了，有很多章节直接跳过了。
- 何登成的github：https://github.com/hedengcheng/tech (https://github.com/hedengcheng/tech)

  登博的博客非常好，对事务、优化这讨论的细节也非常多，不过由于大多是PPT结构，字太少，对上下文不清楚的同学可能会一脸懵逼。
- orczhou的博客：http://www.orczhou.com/ (http://www.orczhou.com/)
- Jeremy Cole的博客：https://blog.jcole.us/innodb/ (https://blog.jcole.us/innodb/)

Jeremy Cole大神不仅写作了 innodb_ruby 这个非常棒的解析 InnoDB 存储结构的工具，还对这些存储结构写了一系列的博客，在我几乎要放弃深入研究表空间结构的时候，是他老人家的博客把我又从深渊里拉了回来。

- 那海蓝蓝（李海翔）的博客：https://blog.csdn.net/fly2nn (https://blog.csdn.net/fly2nn)
- taobao月报：http://mysql.taobao.org/monthly/ (http://mysql.taobao.org/monthly/)

  因为MySQL的源码非常多，经常让大家无从下手，而taobao月报就是一个非常好的源码阅读指南。

  > 吐槽一下，这个taobao月报也只能当作源码阅读指南看，如果真的不看源码光看月报，那只能当作天书看，十有八九被绕进去出不来了。

- MySQL Server Blog：http://mysqlserverteam.com/ (http://mysqlserverteam.com/)

  MySQL team的博客，一手资料，在我不知道看什么的时候给了很多启示。
- mysql_lover的博客：https://blog.csdn.net/mysql_lover/ (https://blog.csdn.net/mysql_lover/)
- Jørgen's point of view：https://jorgenloland.blogspot.com/ (https://jorgenloland.blogspot.com/)
- mariadb的关于查询优化的文档：https://mariadb.com/kb/en/library/query-optimizations/ (https://mariadb.com/kb/en/library/query-optimizations/)

  不得不说mariadb的文档相比MySQL的来说就非常有艺术性了（里边儿有很多漂亮的插图），我很怀疑MySQL文档是程序员直接写的，mariadb的文档是产品经理写的。当我们想研究某个功能的原理，在MySQL文档干巴巴的说明中找不到头脑时，可以参考一下mariadb娓娓道来的风格。
- Reconstructing Data Manipulation Queries from Redo Logs：https://www.sba-research.org/wp-content/uploads/publications/WSDF2012_InnoDB.pdf (https://www.sba-research.org/wp-content/uploads/publications/WSDF2012_InnoDB.pdf)
- 关于InnoDB事务的一个PPT：https://mariadb.org/wp-content/uploads/2018/02/Deep-Dive_-InnoDB-Transactions-and-Write-Paths.pdf (https://mariadb.org/wp-content/uploads/2018/02/Deep-Dive_-InnoDB-Transactions-and-Write-Paths.pdf)
- 非官方优化文档：http://www.unofficialmysqlguide.com/optimizer-trace.html (http://www.unofficialmysqlguide.com/optimizer-trace.html)

  这个文档非常好，非常非常好~
- MySQL8.0的源码文档：https://dev.mysql.com/doc/dev/mysql-server (https://dev.mysql.com/doc/dev/mysql-server)

26.1.2 一些书籍

- 《数据库查询优化器的艺术》李海翔著

  大家可以把这本书当作源码观看指南来看，不过讲的是5.6的源码，5.7里重构了一些，不过大体的思路还是可以参考的。
- 《MySQL运维内参》周彦伟、王竹峰、强昌金著

  内参里有许多代码细节，是一个阅读源码的比较好的指南。
- 《Effectiv MySQL：Optimizing SQL Statements》Ronald Bradford著

  小册子，可以一口气看完，对了解MySQL查询优化的大概内容还是有些好处滴。
- 《高性能MySQL》瓦茨 (Baron Schwartz) / 扎伊采夫 (Peter Zaitsev) / 特卡琴科 (Vadim Tkachenko) 著

  经典，对于第三版的内容来说，如果把第2章和第3章的内容放到最后就更好了。不过作者更愿意把MySQL当作一个黑盒去讲述，主要是说明了如何更好的使用MySQL这个软件，这一点从第二版向第三版的转变上就可以看出来，第二版中涉及的许多的底层细节都在第三版中移除了。总而言之它是MySQL进阶的一个非常好的入门读物。
- 《数据库事务处理的艺术》李海翔著

同《数据库查询优化器的艺术》。

- 《MySQL技术内幕：InnoDB存储引擎 第2版》姜承尧著

  学习MySQL内核进阶阅读的第一本书。

- 《MySQL技术内幕 第5版》 Paul DuBois 著

  这本书是对于MySQL使用层面的一个非常详细的介绍，也就是说它并不涉及MySQL的任何内核原理，甚至连索引结构都懒得讲。像是一个老妈子在给你不停的唠叨吃饭怎么吃，喝水怎么喝，怎么上厕所的各种絮叨。整体风格比较像MySQL的官方文档，如果有想从使用层面从头了解MySQL的同学可以尝试的看看。

- 《数据库系统概念》（美）Abraham Silberschatz / （美）Henry F.Korth / （美）S.Sudarshan 著

  这本书对于入门数据库原理来说非常好，不过看起来学术气味比较大一些，毕竟是一本正经的教科书，里边有不少的公式啥的。

- 《事务处理 概念与技术》Jim Gray / Andreas Reuter 著

  这本书只是象征性的看了1～5章，说实话看不太懂，总是get不到作者要表达的点。不过听说业界非常推崇这本书，而恰巧我也看过一点，就写上了，有兴趣的同学可以去看看。

26.1.3 说点不好的

上边尽说这些参考资料如何如何好了，主要是因为在我写作过程中的确参考到了，没有这些资料可能三五年都无法把小册写完。但是除了MySQL的文档以及《高性能MySQL》、《Effectiv MySQL：Optimizing SQL Statements》这两本书之外，其余的资料在大部分时间都是看的我头晕眼花，四肢乏力，不看个十遍八遍基本无法理清楚作者想要表达的点，这也是我写本小册的初衷---让天下没有难学的知识。

26.1.4 结语

希望这是各位2019年最爽的一次知识付费，如果各位因为阅读本小册而顺利通过面试，或者解决了工作中的很多技术问题，觉得29.9实在是太物超所值，希望各位能来给点打赏（本人很穷，靠救济生活～ 添加好友可以问关于小册的问题，不过希望不要扯犊子聊八卦了，我其实挺忙的～ 微信号：xiaohaizi4919）。

> 小贴士：
> 请允许我鄙视一下那些打着知识付费骗钱的人，除了不生产一点社会价值外，反而生产了数不清的焦虑，让人们连幸福感都丧失掉了。也请各位警惕那些说只要你交几百块钱，就能得到诸如境界上的提升、开阔了眼界、追赶上行业发展趋势之类的课程/知识付费，这类抽象而无法验证的主题都是骗人的。